SWE-DUEL: TOWARDS A SELF-SCALING ARENA FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Static coding benchmarks are fixed at release. Over time their tasks leak into training data, and stronger agents eventually saturate them. We argue that a benchmark should instead aim to be self-scaling: its tasks and oracles are created at evaluation time, so no content exists in advance to leak, and its difficulty is set by the participating agents rather than fixed at release. We introduce SWE-DUEL, an arena in which two coding agents evaluate each other over mirrored Red–Blue turns. In each turn, the Red agent works on a real repository across three context-isolated sessions. It proposes and implements a new feature with feature tests, then embeds a subtle bug in that feature along with sealed bug-triggering tests. The resulting pull request is admitted only if (1) it passes the feature tests and the repository's existing tests while failing exactly the bug tests, and (2) a fresh session of the same agent can repair the bug blind, which rules out unsolvable tasks. The Blue agent then reviews the pull request without being told a bug exists. It wins only if its revision passes all three test suites, and then the two agents swap roles. We run a tournament among 11 model–harness participants on 15 repositories in five languages for 1,500 turns. The leaders win in different ways. GPT-5.5 with Codex CLI ranks first as the strongest solver, repairing 98.3% of the defects it reviews. Claude Opus 4.8 with mini-swe-agent ranks second as the strongest task author, with a 27.3% Red win rate. Finally, within one model family, tasks written by the newest version defeat a fixed-reference solver far more often than those of older versions (42.9% vs. 4.0%), giving preliminary evidence that task difficulty scales with task authoring capability.

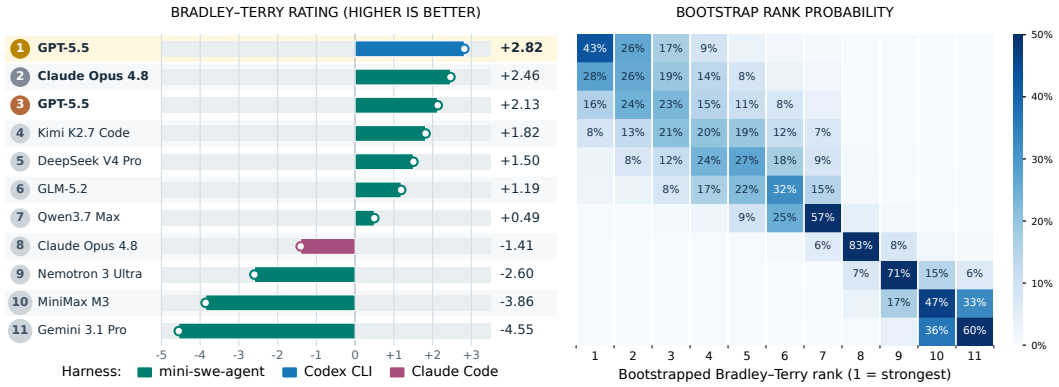


Figure 1: **SWE-Duel Tournament Rankings after a round-robin tournament involving 11 participants.** Gemini 3.1 Pro with mini-swe-agent is a late entrant, demonstrating that new LLMs or harnesses can be evaluated without discarding the current tournament.

1 INTRODUCTION

Benchmarks are the primary instrument for measuring progress in coding agents. To deliver trustworthy signal over the long run, a benchmark must remain absent from the training data of the models it evaluates and challenging for agents to solve. This is increasingly difficult to satisfy, as LLMs

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.

continually learn from publicly available materials and grow more capable with every generation. The deeper issue is that most existing coding benchmarks are static once released and built from openly accessible sources. Such content is eventually absorbed into training corpora, and a fixed task set is eventually saturated by stronger agents. In other words, from the moment a benchmark is released, it begins to leak into training data and to fall behind the agents it is meant to measure. We argue that avoiding this consequence requires a benchmark to be *self-scaling*: its tasks and oracles are created at evaluation time, so no content exists in advance to leak, and its difficulty is set by the participating agents rather than fixed at release, so it can rise as those agents improve.

Existing benchmarks fall short of this property in different ways. Static benchmarks have already run into this problem: OpenAI recently stopped reporting results on SWE-bench Verified (Jimenez et al., 2024) after finding that frontier models could reproduce its gold patches. Continuously updated benchmarks such as SWE-bench-Live (Zhang et al., 2025) and SWE-rebench (Badertdinov et al., 2025) mitigate leakage by collecting tasks created after model training cutoffs, but their task difficulty is still bounded by the daily issues faced by human developers. Most recently, model-generated tasks have emerged as a new direction: BugPilot (Sonwane et al., 2025) collects bugs introduced unintentionally by agents implementing new features, and SwingArena (Xu et al., 2026) pairs LLMs as patch submitters and test-writing reviewers in an adversarial arena. However, BugPilot generates its tasks ahead of time as a fixed set and SwingArena still draws tasks from a fixed pool of human-written GitHub issues, generating only the oracles at evaluation time; neither produces tasks whose difficulty grows with the agent being evaluated. This motivates the central question of this work: **Can we create a self-scaling benchmark for coding agents?**

To answer this question, we present SWE-DUEL, a self-scaling arena with a evaluation protocol in which two coding agents evaluate each other over mirrored adversarial turns. In each turn, the Red agent, working on a given repository across three context-isolated sessions, proposes a new feature, implements it together with feature tests, and finally injects subtle bugs along with bug-triggering tests. The resulting PR is used in the duel only if 1) it passes the feature tests and the repository's regression tests while failing exactly the bug-triggering tests, and 2) the Red agent can repair its own bugs in a fresh session, which rules out unsolvable tasks. The Blue agent then reviews the PR and submits a revision, winning the round if and only if the revision passes all three test suites; the two agents swap roles in the next turn. Because every feature, bug, and oracle is created at evaluation time, these specific tasks cannot appear in earlier training data, And because tasks are authored by the agents under evaluation, task difficulty is not fixed by a human-written pool; it can in principle rise as the authors improve, up to the limit set by the self-repair gate. Altogether, a single duel profiles a broad slice of the software engineering lifecycle, including requirement specification, feature implementation, test generation, code review, fault localization, and program repair.

We evaluate SWE-DUEL in a tournament among 11 participants, each pairing a frontier proprietary or open-weight model (e.g., GPT-5.5, Claude Opus 4.8, DeepSeek V4 Pro, and Gemini 3.1 Pro) with one of three agent harnesses, on 15 repositories in five languages. Ten participants play a full round robin, and Gemini 3.1 Pro joins afterwards as a late entrant, giving 50 matches and 1,500 turns. The leaders win in different ways. GPT-5.5 with Codex CLI ranks first as the strongest solver, repairing 98.3% of the defects it reviews, while Claude Opus 4.8 with mini-swe-agent ranks second as the strongest task author, with a 27.3% Red win rate. A late entrant can be placed with only five matches, and within one model family, tasks written by the newest version defeat a fixed reviewer far more often than those of older versions (42.9% vs. 4.0%), giving preliminary evidence that difficulty tracks authoring capability.

We make the following contributions:

- We introduce SWE-DUEL, a self-scaling arena that evaluates coding agents with each other. It does not leak test content by design and determines the task difficulty with the participants' own authoring ability rather than fixed at release.
- We build an open arena toolkit for SWE-DUEL, enabling the community to evaluate any new agent on freshly generated tasks (Anonymous, 2026b).
- We conduct a 1,500-turn tournament over 11 (model, harness) pairs, demonstrating the broad applicability of SWE-DUEL and revealing distinct behavioral profiles across models and harnesses.

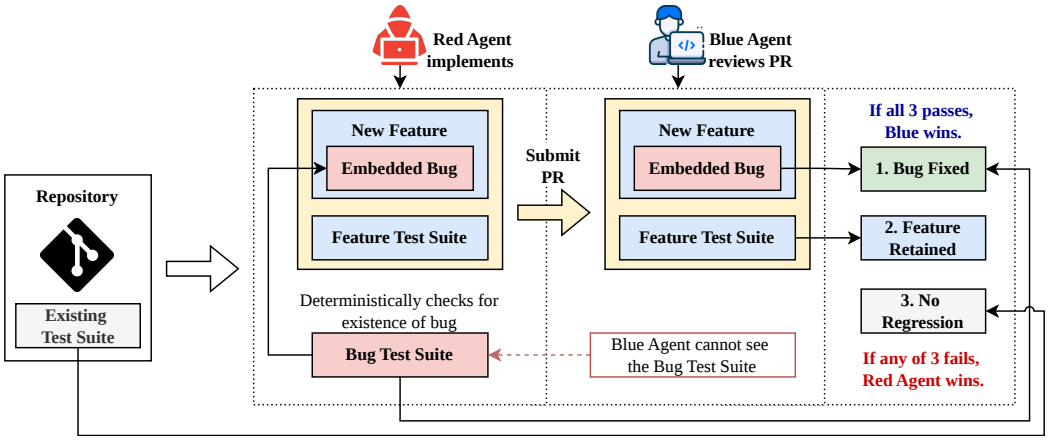


Figure 2: **A contamination-resistant evaluation in SWE-DUEL.** Red agent implements a new feature with an embedded bug, so that the Blue agent cannot simply revert the repository to its original version to remove the bug as doing so would remove the implemented feature. Feature retention is deterministically verified with the feature test suite, while bug removal is deterministically verified with the bug test suite. Existing test suite verifies no regression by the Blue agent's patch.

2 RELATED WORK

Repository-level coding benchmarks. SWE-bench (Jimenez et al., 2024; OpenAI, 2024) moved evaluation from isolated functions to real issue resolution. Later benchmarks add automated environment setup (Vergopoulos et al., 2025), more languages (Rashid et al., 2025), long-horizon software evolution (Le et al., 2026; Chen et al., 2026), the full issue-resolution cycle (Guan et al., 2026b), and feature addition (Liu et al., 2026). In all of them, the tasks and reference patches are public and fixed before evaluation. However, public tasks eventually enter training data of evaluated LLMs (Matton et al., 2024). The benchmarks thus need to be repeatedly updated to replace the leaked ones. For example, SWE-bench-Live (Zhang et al., 2025) and SWE-rebench (Badertdinov et al., 2025) regularly collect new tasks from GitHub to remain temporally fresh. (Guan et al., 2026a) and DyCodeEval (Chen et al., 2025) propose to perturb the tasks to avoid exact memorization, but preserve the original tasks and solutions. Moreover, the tasks come from human development history (issues, pull requests, and releases), so their difficulty is fixed at release, and a fixed task set is saturated as agents improve. The usual response is to build a harder benchmark, such as SWE-Bench Pro (Deng et al., 2025), but each replacement again has a fixed difficulty. Refreshing the benchmark does not solve this as the new tasks come from the same distribution, so the tasks are newer but not harder. In SWE-Duel, both the tasks and their oracles are written at evaluation time by the agents being evaluated. No task exists in advance to leak, and difficulty is set by the participants instead of a fixed human-written pool.

Automatic bug generation. Early work injects bugs with predefined or learned patterns: symbolic synthesis of deep faults (Roy et al., 2018), bug seeding learned from past fixes (Patra & Pradel, 2021), learned contextual mutations (Richter & Wehrheim, 2022), and re-introducing fixed security bugs (Zhang & Patterson, 2022). These methods show that realistic bugs can be generated automatically, but each bug is a local edit drawn from a pattern set, so the variety of bugs is limited by the patterns. LLMs remove this limit. SWE-smith (Yang et al., 2025) prompts LLMs to modify or rewrite functions, BugPilot (Sonwane et al., 2025) collects the regressions that agents cause by accident while adding features, and Self-play SWE-RL (Wei et al., 2025) trains one model to both inject and repair bugs. Both target training data, and they show that LLMs can produce diverse and hard bugs at scale. BugPilot is the closest to our task construction, since its bugs also arise while an agent adds a feature. The key difference is what the solver can see. A BugPilot bug is kept only if it breaks existing tests, so running the test suite reveals it. A SWE-DUEL bug is admitted only if all visible tests still pass, so the solver has to find it by reading the code, and only a sealed test can confirm the fix. Our bugs are also deliberate, so the task yield does not depend on agents making mistakes, which may become rarer as agents get more reliable. For evaluation, CDBench (Romazanov et al., 2026) lets an attacker LLM write mutants for a defender LLM, but the mutants are local edits and

the defender is shown the mutant diff. In SWE-DUEL, each agent hides a bug inside a new feature for the other to fix, and a task is admitted only after a fresh session of its author repairs the bug blind, so every task is known to be solvable.

Arena-style evaluation. Chatbot Arena (Chiang et al., 2024) ranks LLMs from pairwise human votes, and Arena-Lite (Son et al., 2025) and Auto-Arena (Zhao et al., 2025) replace human voters with LLM judges and tournament schedules. Arenas stay fresh because new prompts keep arriving, but their outcomes depend on judges, whose preferences are noisy and are not the same as correctness. SwingArena (Xu et al., 2026) brings the arena format to software engineering and lets repository CI decide the outcome: submitters patch GitHub issues and reviewers write tests. However, its tasks come from a fixed pool of human-written issues, and reviewer tests are validated against a human gold patch. SWE-DUEL keeps execution-based scoring but removes both human anchors: the competing agents write the tasks themselves, and task validity is checked by execution gates and a blind self-repair instead of a gold patch. As a result, task difficulty is set by the participants and not by a human-written pool.

3 SWE-DUEL: A SELF-SCALING CODING ARENA

SWE-DUEL evaluates coding agents through Red–Blue duels (Figure 2). A duel is played by two participants on a real repository R, and each participant plays both roles in turn. The ***Red*** side creates a task which is a pull request that adds a new feature to R and hides a bug inside that feature. The ***Blue*** side solves the task by reviewing the pull request and submits a fixed version if the bug is noticed.

Formally, a task is a tuple $C = (R, s, d, T_f, T_b)$, where s is the feature specification, d is the pull-request diff that contains both the feature and the bug, T_f is a feature test suite, and T_b is a bug test suite that fails on d only because of the bug. Blue receives only the public part, $\mathrm{pub}(C) = (R, s, d, T_f)$. T_b stays sealed, so Blue is not told whether or where a bug exists. Blue returns a revised diff d' and wins the turn if and only if the repository's existing tests T_e, the feature tests T_f, and the sealed bug tests T_b all pass on d'; otherwise Red wins. Because the bug lives inside the feature, Blue cannot win by reverting the pull request, since T_f would then fail.

Red's tasks are created and validated before the tournament and stored in a task bank (Section 3.2), so every opponent faces the same tasks from a given Red. In a match, the two participants defend each other's banked tasks on every repository (Section 3.3), and a tournament is a set of matches among many participants (Section 3.4).

3.1 PARTICIPANTS AND ENVIRONMENT

A participant is a pair (m, h), where m is the model and h is the harness that provides the agent loop, tools, context policy, command execution, and recovery behavior. The same model under two harnesses counts as two participants, each with its own task bank and records.

Every duel runs inside a Docker container built for one open-source repository pinned to a specific commit. The repository's existing test suite records the original behavior and is re-run on every later artifact (the new feature, the embedded bug, and Blue's patch) to detect regressions. Red sessions, validation gates, and scored Blue runs all start from the same image, so every participant works in an identical environment.

3.2 RED AGENT: TASK CREATION

Red creates each task in three sessions, and each session is followed by an executable validation gate (Figure 3). Every session starts with a fresh context, so a session knows only what we pass to it. For each participant and repository, we request N tasks, one per *slot*. A slot allows at most five attempts. The first attempt that passes all three gates is admitted and stored in the participant's task bank. If all five attempts fail, the slot is *exhausted* and stays empty. We never replace an exhausted slot with a new one, because creating a valid task (implementing a feature, embedding a bug, writing both test suites, and passing self-repair) is part of the ability we measure. An exhausted slot later counts as a missing task in a match (Section 3.4).

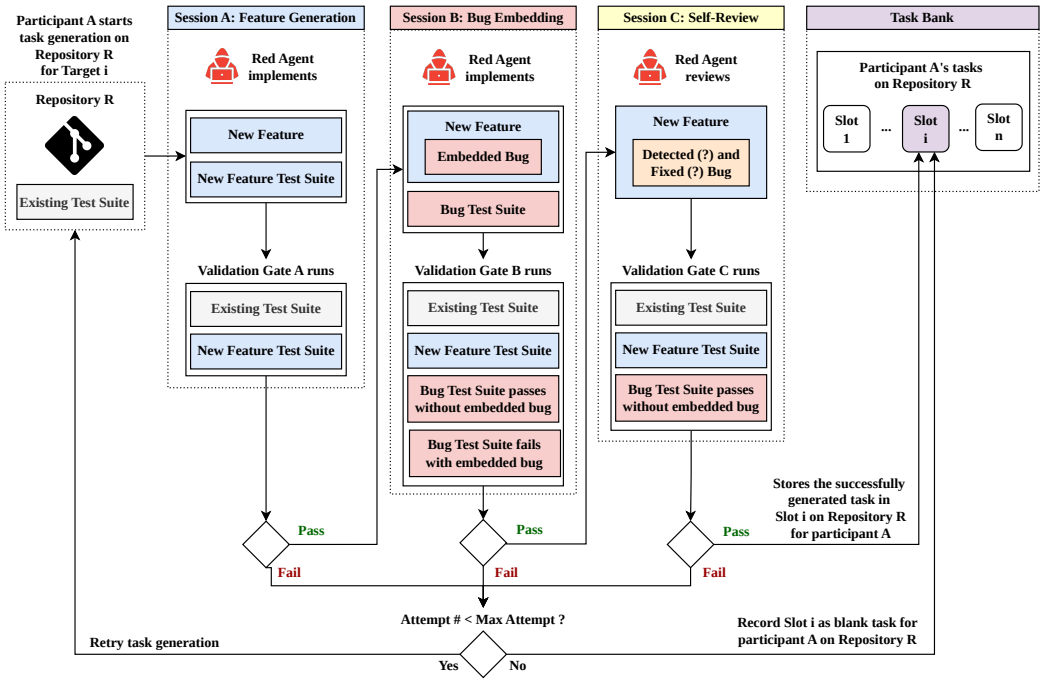


Figure 3: **Three-session task generation.** Each of the n targets is stored inside a slot. Each slot permits at most five attempts: a challenge successfully generated within 5 attempts is stored inside its corresponding slot, whereas if it exceeds five attempts, it leaves an exhausted slot. Validation gate A ensures the verifiability of the implemented feature. Validation gate B ensures the verifiability of the embedded bug. Validation gate C ensures the fairness of the embedded bug by verifying the same-system reviewer detects and removes the bug from the new feature.

Session A: Feature Implementation Red explores the repository R, picks a target module on its own, implements a new feature, and writes the feature specification s and the feature tests T_f. We denote the resulting feature-only diff by d_f. To keep a participant's tasks on one repository diverse, Red is shown the specifications and target files of its earlier admitted tasks there and must pick a different module. The validation gate of session A requires that the existing test suite T_e and generated feature tests T_f pass on d_f and that the diff and the tests meet minimum size thresholds. Tests that already fail on the pinned commit are recorded and ignored. This gate establishes that the feature is correct before any bug is added, so a later failure can be attributed to the bug and not to a broken feature.

Session B: Bug Embedding A fresh Red session receives the feature specification s and the diff d_f. It edits the feature code to embed a bug, which gives the final diff d, and writes the bug tests T_b. The validation gate of session B requires that (i) d applies to the pinned commit and passes lint or compilation; (ii) the build, T_e, T_f, and the size checks still pass on d; and (iii) T_b passes on d_f but fails on d. Condition (ii) ensures that the bug gives Blue no visible failing test. Condition (iii) ensures that T_b tests the bug and not the feature, and that the presence of the bug can be checked by execution.

Session C: Blind Self-Repair Deterministic gates of session A and B show that the embedded bug exists and is hidden, but not that it can be detected and fixed from what a reviewer sees. Therefore, a fresh instance of the same participant receives only the public bundle $\mathrm{pub}(C)$ (i.e., repository, feature specification, pull-request diff, and feature tests), which is exactly what Blue will later receive, and submits a revised diff d'. The task is admitted only if T_e, T_f, and T_b all pass on d', which is the same condition under which Blue wins a turn. This keeps unsolvable tasks out, but it also means a task can be no harder than its author's own bug-finding ability. We view this as a deliberate trade-off between solvability and difficulty. Admitted tasks are saved in a bank and reused in every match of a tournament, so all opponents face the same tasks from a given Red.

3.3 BLUE AGENT: TASK SOLVING

Blue receives the public bundle $\mathrm{pub}(C)$ of a task C from Red's bank, exactly as the reviewer in Session C does (Section 3.2). The bug tests T_b and the bug metadata stay hidden. Blue starts from a fresh context in the same container image and is asked to review the pull request as if it came from a contributor. The prompt does not say that a bug exists. Blue may edit the code, including rewriting the feature, or leave the pull request as it is, and then submits its final diff d'.

The framework then runs the three test suites on d'. Let $S_e, S_f, S_b \in \{0, 1\}$ indicate that T_e, T_f, and T_b pass. We deliberately use a hard conjunction:

$$S_{\text{Blue}} = S_e S_f S_b, \qquad S_{\text{Red}} = 1 - S_{\text{Blue}}. \tag{1}$$

Blue wins the turn only if all three suites pass. There is no partial credit for a plausible but incomplete patch. Blue can lose in three ways: the bug ships ($S_b = 0$), the feature is broken or removed ($S_f = 0$), or existing behavior regresses ($S_e = 0$). If Blue submits the pull request unchanged, Gate B guarantees that T_b fails, so Red wins.

3.4 TOURNAMENT

A match between participants A and B is played over a set of repositories $\mathcal{R} = \{R_1, \ldots, R_{|\mathcal{R}|}\}$ with N tasks per repository. Let $C_{c,j,i}$ denote the i-th admitted task of participant c on repository R_j. For every R_j and every $i \leq N$, the match has two turns: B plays Blue on $C_{A,j,i}$, and A plays Blue on $C_{B,j,i}$. A match therefore has $2N|\mathcal{R}|$ turns, and each participant plays Red and Blue equally often on the same repositories. If a participant admitted fewer than N tasks on R_j because its slots were exhausted, each missing task is scored as a Blue win without running Blue. We call all other turns *contested*. Missing tasks thus lower their author's match score. Each turn gives one point to Blue if $S_{\text{Blue}} = 1$ (Eq. 1) and to Red otherwise. A participant's match score Q is the sum of its points in both roles. A wins the match if $Q_A > Q_B$, loses if $Q_B > Q_A$, and draws otherwise.

Following Chatbot Arena (Chiang et al., 2024) and Arena-Lite (Son et al., 2025), SWE-DUEL uses a round-robin format in which every pair of participants plays one match. To add a new participant to a finished tournament of M participants, the new participant first creates its own task bank and then plays against only $\lceil \log_2 M \rceil + 1$ incumbents, selected with the active sampling rule of Chatbot Arena (Chiang et al., 2024). The incumbents' task banks and all earlier match results are reused unchanged, so the number of new matches grows logarithmically with M.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Repository field. SWE-DUEL uses 15 popular repositories across 5 programming languages (Python, Java, Go, JavaScript, C/C++) as shown in Table 4 of Appendix C.

Agents. We choose frontier models at the time of our experiment, including GPT-5.5 (OpenAI, 2026a), Claude Opus 4.8 (Anthropic, 2026a), Kimi K2.7 Code (Kimi, 2026), DeepSeek V4 Pro (DeepSeek-AI et al., 2026), GLM 5.2 (Z.ai, 2026), Qwen 3.7 Max (QwenTeam, 2026), Nvidia Nemotron 3 Ultra (NVIDIA et al., 2026), Minimax M3 (MiniMax, 2026), and Gemini 3.1 Pro (Google, 2026). All these models are paired with mini-swe-agent (Yang et al., 2024) as harness. In addition, we include two more (model, harness) pairs: GPT-5.5 with Codex CLI (OpenAI, 2026b) and Claude Opus 4.8 with Claude Code (Anthropic, 2026b). Step budgets, recovery turns, decoding temperature, and the execution environment are detailed in Appendix D.1. All model APIs are accessed through OpenRouter between June 18 and August 30, 2026; per-model access dates are reported in Appendix D.2.

Tournament Details. In our experiments, we start with an original group of ten participants. To minimize variance in rankings, each of the original ten participants faced every other participant in a round-robin format, forming $\binom{10}{2} = 45$ match pairings. After the initial round-robin tournament, we demonstrate the late entrance by Gemini 3.1 Pro which plays 5 active-sampled incumbents, giving $45 + 5 = 50$ matches. Since, we use 15 repositories in SWE-DUEL, we have $15 \times 2 = 30$ turns in a match. Thus, in total, we have 50 matches $\times$ 30 turns per match $= 1{,}500$ turns. We give automatic

Table 1: **Main tournament results.** The rows are sorted in descending order by the full-schedule Bradley-Terry log-strength rating in the highlighted column. Full-schedule means the missing-Red auto-wins are retained. Figure 5 shows the trajectory of the rankings throughout round-robin tournament rounds before arriving at this final ranking.

Competitor	Matches played	Won	Drawn	Lost	Red win (%)	Blue win (%)	Bradley–Terry
GPT-5.5 [codex]	10	8	1	1	11.3	98.7	**+2.82**
Claude Opus 4.8 [mini-swe-agent]	10	8	0	2	27.3	88.7	**+2.46**
GPT-5.5 [mini-swe-agent]	9	5	3	1	10.4	97.0	**+2.13**
Kimi-K2.7-Code [mini-swe-agent]	9	5	2	2	15.6	90.4	**+1.82**
DeepSeek-V4-Pro [mini-swe-agent]	9	5	1	3	18.5	91.9	**+1.50**
GLM-5.2 [mini-swe-agent]	9	4	2	3	14.8	89.6	**+1.19**
Qwen3.7-Max [mini-swe-agent]	9	3	2	4	13.3	91.1	**+0.49**
Claude Opus 4.8 [claude-code]	10	2	2	6	7.3	87.3	**-1.41**
Nemotron-3-Ultra [mini-swe-agent]	10	1	2	7	6.7	79.3	**-2.60**
MiniMax-M3 [mini-swe-agent]	10	1	0	9	2.0	74.7	**-3.86**
Gemini-3.1-Pro [mini-swe-agent]	5	0	1	4	2.7	72.0	**-4.55**

win to *Blue* on missing *Red* challenges so as to avoid survivor bias (dropping weak task creators), and also to consider task creation as SWE capability of the Red agent, not just task solving. For completeness, we also report the alternative tournament ranking without automatic win for *Blue* on missing *Red* challenges, to include only the contested turns.

Metric. Following existing works in arena-style evaluation of LLMs (Chiang et al., 2024; Zhao et al., 2025; Son et al., 2025), SWE-DUEL reports tournament outcome using **Bradley–Terry (BT) ratings** (Bradley & Terry, 1952). It assigns each participant p a log-strength β_p (centered to zero mean), fitted by maximum likelihood over all match outcomes of Section 3.4 under the model $\Pr(p \text{ beats } q) = \sigma(\beta_p - \beta_q)$ with σ the logistic function and a draw counted as half a win for each side, from which we report rank probability distributions and 95% confidence intervals using 1,000 bootstrap resamples of matches.

4.2 MAIN RESULTS

Overall ranking. Figure 1 depicts the tournament rankings of all 11 agents in BT ratings, and the probability distribution of the rankings based on the participants' BT ratings. Meanwhile, Table 1 reports definitive full-schedule BT ranking together with comprehensive statistics such as Win/Loss/Draw, and Bradley–Terry ratings. GPT-5.5 with Codex harness leads the final ranking in BT ratings, followed in order by Claude Opus 4.8, Kimi-K2.7 Code, and DeepSeek-V4-Pro, all with mini-swe-agent harness. Meanwhile, MiniMax-M3 and the late entrant Gemini 3.1 Pro rank the lowest. Table 6 presents 95% confidence intervals for the rankings and BT ratings. The six participants with strongest BT ratings (GPT-5.5 under both harnesses, Opus 4.8 with mini-swe-agent, Kimi, DeepSeek, and GLM) form a broad leading group, whereas the lower group comprises Opus 4.8 with Claude Code, Nemotron and MiniMax, and the five-match Gemini late entrant. Appendix N reports the complementary turn-level BT refit of the main tournament.

Distinct profiles in creating and solving tasks. In Figure 8, we report normalized, challenge-clustered role rates. The two leaders reach the top in very different ways. GPT-5.5 is the strongest solver under both harnesses, with Blue win rates of 98.7% with Codex CLI and 97.0% with mini-swe-agent. In contrast, Claude Opus 4.8 with mini-swe-agent is the strongest task creator: its Red win rate of 27.3% is the highest in the field and more than twice that of either GPT-5.5 participant (11.3% and 10.4%), which suggests that it embeds bugs that are harder to find. DeepSeek-V4-Pro's Red:Blue record of 18.5%–91.9% is seemingly better than that of Kimi-K2.7-Code (15.6%:90.4%). Yet, Kimi ranks above DeepSeek on the BT ratings (with 95% CIs overlapping) due to Kimi having one less loss than DeepSeek. At the lower end, MiniMax-M3 (2.0%) and the Gemini 3.1 Pro (2.7%) have the weakest task creation capabilities, thus often failing to generate admitted tasks, and thereby give the Blue opponent an automatic win on the task solving turn. Appendix L and Appendix M ablates the automatic wins, and analyzes the two capabilities separately.

Table 2: Fixed-reference solver failure rates across the task-author capability ladder. Successive versions of one model family (all with mini-swe-agent) each authored tasks over the same repositories. The 13 GLM-5.2 draws are the solver's in-match tournament solving runs reused verbatim; the other 26 ran fresh with zero infrastructure errors.

Task author	Tasks	Yield	Solver failure (95% Wilson)
GLM-5.1	12	30.8%	$1/12 = 8.3\%$ $[1.5, 35.4]$
GLM-5.2	13	40.6%	$0/13 = 0.0\%$ $[0.0, 22.8]$
GLM-5.3	14	37.8%	$6/14 = 42.9\%$ $[21.4, 67.4]$
All versions	39	36.1%	$7/39 = 17.9\%$ $[9.0, 32.7]$

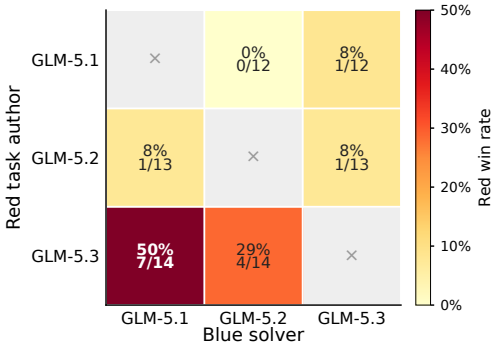


Figure 4: Head-to-head match-ups between the ladder versions (all with mini-swe-agent).

Task solving rate and its breakdown. Across all 1,500 recorded turns, Blue succeeds 87.9% of the time (95% CI $[84.6\%, 90.9\%]$). Focusing on the 1,219 contested turns, Blue succeeds 85.1% (95% CI $[81.0\%, 88.6\%]$): 1,037 clean repairs, 158 cases where the latent bug ships, and 24 collapses that break the feature or existing behavior. Figure 10 in Appendix P breaks contested success down by programming language and by bug category, with 95% task-clustered intervals. By language, Blue success is 89.2% for Java, 85.7% for Go, 83.9% for C/C++, 83.5% for Python, and 82.9% for Node. By self-reported bug category, logic errors are the hardest to repair (81.9%), followed by security vulnerabilities (84.9%), off-by-one errors (88.9%), type confusion (90.6%), and input validation (95.5%).

Adding a new participant. To demonstrate how a new agent joins the existing leaderboard, we held Gemini 3.1 Pro with mini-swe-agent out of the ten-player round robin and added it only after all 45 incumbent matches had finished. The newcomer first created its own task bank over the same 15 repositories under the same protocol. SWE-DUEL then scheduled $\lceil \log_2 10 \rceil + 1 = 5$ matches against incumbents chosen by active sampling, instead of all ten pairings. The incumbents' task banks and earlier match results were reused unchanged. Five matches were enough for a coherent placement. Gemini finished 0–1–4 (W/D/L) and ranks last in BT rating (Table 1), with the lowest contested Blue success in the field (61.8%, Figure 8). Adding it also left the existing leaderboard untouched: refitting Bradley–Terry on the 45 incumbent-only matches reproduces the incumbent ordering of the full 50-match fit exactly (Kendall's $\tau = 1.0$). Appendix S details the onboarding protocol.

Tournament and newcomer cost. Running this tournament consumed 2,109 USD of API spend: $670 for Red task creation (including every failed attempt), $158 for blind self-review validation, and $1,281 for Blue evaluation, about $26 per match. Per-participant totals are in Table 12 of Appendix R. Onboarding the newcomer cost $243: $126 to author its task bank once, plus $117 for its five matches, split between the entrant solving incumbent tasks ($61.81) and incumbents solving the entrant's tasks ($55.64). A newcomer's marginal cost is therefore its own bank plus $O(\log N)$ active-sampled matches rather than the $O(N)$ pairings of a full round robin: had Gemini 3.1 Pro played all ten incumbents, we estimate the entrance at about $379 (Appendix S).

4.3 ADDITIONAL ANALYSIS

Adaptive task difficulty. SWE-Duel assumes that stronger agents create harder tasks. The tournament cannot test this directly, because participants differ in more than strength. We therefore fixed the solver and changed only the task author: GLM-5.1, GLM-5.2, and GLM-5.3 (all with mini-swe-agent) each wrote tasks on the same repositories, and DeepSeek-V4-Pro × mini-swe-agent, a reviewer outside the family, solves all admitted tasks. Table 2 shows the fixed-reference solver's failure rates when solving tasks from the three task authors. It fails on 8.3% (GLM-5.1), 0.0% (GLM-5.2), and 42.9% (GLM-5.3) of them (exact trend test p=.015). It suggests that the tasks from two older versions are both too easy for DeepSeek-V4-Pro, while the newest is clearly harder. We also ran a round-robin tournament among GLM-5.1, GLM-5.2, and GLM-5.3 that allows us to present head-to-head match-ups that further highlight the models' different capabilities (Figure 4).

Table 3: Fixed reference task solver over the tasks used in the main tournament. The endogenous Red win rate is its Red success over its contested-only tournament turns; the reference Red win rate is the fraction of its admitted tasks that the fixed reference solver fails to solve blind.

Participant	n tasks	Endog. Red wins	Endog. rate	Ref. Red wins/n	Ref. rate (95% Wilson)
Claude Opus 4.8 [mini-swe-agent]	15	41/150	27.3%	4/15	26.7% [10.9, 52.0]
DeepSeek-V4-Pro [mini-swe-agent]	15	25/135	18.5%	3/15	20.0% [7.0, 45.2]
Qwen3.7-Max [mini-swe-agent]	15	18/135	13.3%	3/15	20.0% [7.0, 45.2]
Nemotron-3-Ultra [mini-swe-agent]	7	10/70	14.3%	1/7	14.3% [2.6, 51.3]
GLM-5.2 [mini-swe-agent]	13	20/117	17.1%	1/13	7.7% [1.4, 33.3]
GPT-5.5 [Codex CLI]	13	17/130	13.1%	1/13	7.7% [1.4, 33.3]
GPT-5.5 [mini-swe-agent]	13	14/117	12.0%	1/13	7.7% [1.4, 33.3]
Claude Opus 4.8 [Claude Code]	15	11/150	7.3%	1/15	6.7% [1.2, 29.8]
Kimi-K2.7-Code [mini-swe-agent]	13	21/117	17.9%	0/13	0.0% [0.0, 22.8]
MiniMax-M3 [mini-swe-agent]	5	3/50	6.0%	0/5	0.0% [0.0, 43.4]
Gemini-3.1-Pro [mini-swe-agent]	10	2/48	4.2%	0/10	0.0% [0.0, 27.8]
All participants	**134**	–	–	**15/134**	**11.2%** [6.9, 17.6]

GLM-5.3's tasks defeat GLM-5.1 in 7/14 (50.0%) and GLM-5.2 in 4/14 (28.6%) contested turns, while the two older versions only defeats GLM-5.3 once each: 1/12 for GLM-5.1, and 1/13 for GLM-5.2. Appendix K discusses the adaptive task difficulty experiment results in detail.

Fixed-reference task solver. In the tournament, a participant's Red win rate depends on which Blue it happened to meet. To put all participants on one scale, we had a single mid-tier agent (GLM-5.2 with mini-swe-agent) solve all 134 tournament tasks (Table 3). It fails on 15 of 134 tasks (11.2%), and the Red win rates against it correlate with the tournament rates (Spearman ρ=0.66) . Claude Opus 4.8 with mini-swe-agent remains the strongest task creator (27.3% in the tournament, 26.7% against the reference). Kimi-K2.7-Code drops from 17.9% to 0 of 13, which suggests its tournament wins came from a few weaker defenders. Further details about this experiment are in Appendix J.

Admission stability. A task is admitted after a single blind self-repair run (Section 3.2), and that run is stochastic, so some tasks may have been admitted by luck. To measure this, we re-ran the same check three more times on a stratified sample of 44 tournament tasks (4 per participant). Of these, 35 were self-repaired in all three runs, 8 in two, and one in none. The pooled self-repair rate is 91.7% (95% Wilson CI 85.7–95.3%). Admitted tasks are therefore mostly reliably solvable, and lucky admissions are rare (Appendix I).

Bug test case soundness. The sealed bug tests are written by Red, so a test that is too specific could incorrectly reject a legitimate repair. We manually checked a seeded random sample of 50 of the 158 turns that Blue lost only on the bug tests. We found 45 genuine misses, three valid repairs rejected by the test, and two ambiguous cases: a 6.0% false-negative rate (95% Wilson CI 2.1–16.2%). All three false negatives come from two Go JWT tasks whose tests require one specific error type or option (more detail in Appendix G). Five language-stratified examples of challenges are in Appendix B.

Ablated tournament for harness. To investigate the effect of harnesses on the participants, we run an ablated tournament in which the same model (DeepSeek-V4-Pro, GLM-5.2, or Kimi-K2.7-Code) plays against itself under mini-swe-agent, Codex CLI, and Claude Code. Just like the main tournament, this ablation tournament covers the same 15 open-source repositories. Unlike the main tournament, there are no native pairings that could confound the harness effect analysis. Mini-swe-agent leads with BT [95% CI] score of $+0.73\,[-0.25, +5.56]$ followed by Codex CLI wtih $-0.25\,[-3.50, +1.34]$, and Claude Code with $-0.49\,[-3.83, +0.65]$ (Appendix Q).

5 Conclusion

We present SWE-Duel, an arena in which coding agents test each other. In a 1,500-turn tournament among 11 participants, GPT-5.5 with Codex CLI was the strongest solver, repairing 98.3% of the defects it reviewed. Claude Opus 4.8 with mini-swe-agent was the strongest task author, with a 27.3% Red win rate. A new participant joined with only five matches without changing the existing order. Within one model family, tasks written by the newest version defeated a fixed solver far more often than those of older versions, which gives early evidence that task difficulty rises with the author's ability.

REFERENCES

Anonymous. SWE-Duel Dataset, 2026a. URL https://osf.io/n7cdj/overview?view_only=afb5b2f1c939469d8c978996e460786d.

Anonymous. SWE-Duel Toolkit, 2026b. URL https://anonymous.4open.science/r/SWE-Duel-C96E.

Anthropic. Introducing Claude Opus 4.8, May 2026a. URL https://www.anthropic.com/news/claude-opus-4-8.

Anthropic. Run Claude Code programmatically, 2026b. URL https://code.claude.com/docs/en/headless.

Arena.ai. LLM Leaderboard - Best Text & Chat AI Models Compared, 2026. URL https://arena.ai/leaderboard/text.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents, November 2025. URL http://arxiv.org/abs/2505.20411. arXiv:2505.20411 [cs.SE].

Ralph Allan Bradley and Milton E. Terry. Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. *Biometrika*, 39(3/4):324–345, 1952. ISSN 0006-3444. doi: 10.2307/2334029. URL https://www.jstor.org/stable/2334029.

Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. SWE-CI: Evaluating Agent Capabilities in Maintaining Codebases via Continuous Integration, April 2026. URL http://arxiv.org/abs/2603.03823. arXiv:2603.03823 [cs.SE].

Simin Chen, Pranav Pusarla, and Baishakhi Ray. Dynamic Benchmarking of Reasoning Capabilities in Code Large Language Models Under Data Contamination, June 2025. URL http://arxiv.org/abs/2503.04149. arXiv:2503.04149 [cs.SE].

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference, March 2024. URL http://arxiv.org/abs/2403.04132. arXiv:2403.04132 [cs.AI].

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J. Yang, J. Q. Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, M. S. Di, M. Y. Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, S. H. Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu

Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W. Zhang, W. L. Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, Y. C. Yan, Y. Q. Wang, Y. W. Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, Z. F. Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence, April 2026. URL http://arxiv.org/abs/2606.19348. arXiv:2606.19348 [cs.CL].

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?, November 2025. URL http://arxiv.org/abs/2509.16941. arXiv:2509.16941 [cs.SE].

designarena.ai. Design Arena | Leaderboards, 2026. URL https://designarena.ai/leaderboard.

Google. Gemini 3.1 Pro, February 2026. URL https://deepmind.google/models/gemini/pro/.

Batu Guan, Xiao Wu, Yuanyuan Yuan, and Shaohua Li. Is Your Benchmark Still Useful? Dynamic Benchmarking for Code Language Models, July 2026a. URL http://arxiv.org/abs/2503.06643. arXiv:2503.06643 [cs.SE].

Hao Guan, Lingyue Fu, Shao Zhang, Yaoming Zhu, Kangning Zhang, Lin Qiu, Xunliang Cai, Xuezhi Cao, Weiwen Liu, Weinan Zhang, and Yong Yu. SWE-Cycle: Benchmarking Code Agents across the Complete Issue Resolution Cycle, May 2026b. URL http://arxiv.org/abs/2605.13139. arXiv:2605.13139 [cs.SE].

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?, November 2024. URL http://arxiv.org/abs/2310.06770. arXiv:2310.06770 [cs.CL].

Kimi. Kimi K2.7 Code: Open-Source Agentic Coding Model, July 2026. URL https://www.kimi.ai/resources/kimi-k2-7-code.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios, May 2026. URL http://arxiv.org/abs/2512.18470. arXiv:2512.18470 [cs.SE].

Shuhan Liu, Zhiyi Zhao, Xing Hu, Kui Liu, Xiaohu Yang, and Xin Xia. RACE-Bench: A Reasoning-Augmented Benchmark for Repository-Level Code Agents on Feature Addition, August 2026. URL http://arxiv.org/abs/2603.26337. arXiv:2603.26337 [cs.SE].

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. On Leakage of Code Generation Evaluation Datasets, October 2024. URL http://arxiv.org/abs/2407.07565. arXiv:2407.07565 [cs.CL].

MiniMax. MiniMax M3: Frontier Coding, 1M Context, Native Multimodality, June 2026. URL https://www.minimax.io/blog/minimax-m3.

NVIDIA, Aaron Blakeman, Aaron Thomas, Aastha Jhunjhunwala, Abhibha Gupta, Abhinav Khattar, Adam Rajfer, Adi Renduchintala, Adil Asif, Aditya Vavre, Adriana Flores Miranda, Ahmad Bilal, Aileen Zaman, Ajay Hotchandani, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, Alex Gronskiy, Alex Kondratenko, Alex Steiner, Alex Ye, Alexander Bukharin, Alexandre Milesi, Ali Taghibakhshi, Alice Gatti, Alisa Liu, Alok Kumar, Amar Phanishayee, Ameya Sunil Mahabaleshwarkar, Amir Klein, Amit Zuker, Amnon Geifman, Anahita Bhiwandiwalla, Ananth Subramaniam, Andrea Santilli, Andrew Fulks, Andrew McHarg, Andrew Tao, Andrii Skliar, Anjulie Agrusa, Ankur Srivastava, Ankur Verma, Anna Shors, Anna Warno, Antoni-Joan Solergibert I. Llaquet, Arham Mehta, Arkadiusz Nowaczynski, Arti Jain, Ashwath Aithal, Ashwin Poojary, Asif Ahamed, Asit Mishra, Asma Kuriparambil Thekkumpate, Atefeh Sohrabizadeh, Avinash Kaur, Avinash Vem, Ayush Dattagupta, Barath Subramaniam Anandan, Bardiya Sadeghi, Ben Lanir, Benedikt Schifferer, Besmira Nushi, Bilal Kartal, Bill Thiede, Bita Darvish Rouhani, Bo Deng, Bob Schatz, Boris Ginsburg, Boxin Wang, Brad Nemire, Brandon Norick, Brian Dang, Brian Westphal, Brian Yu, Brucek Khailany, Bryan Catanzaro, Carlo del Mundo, Caryln Aarish, Chankyu Lee, Chantal Hwang, Charbel Sakr, Charles Wang, Charlie Truong, Chen Cui, Cheng Cheng, Cheng-Ping Hsieh, Chenghao Zhang, Chenhui Deng, Chintan Patel, Chris Alexiuk, Christian Cosgrove, Christian Munley, Christine Harvey, Christopher Parisien, Chunyang Shen, Coco Li, Collin Neale, Cynthia Gao, Cyril Meurillon, Dan Gil, Dan Su, Dan Zhao, Dane Corneil, Daniel Afrimi, Daniel Egert, Daniel Korzekwa, Daniel Lo, Daniel Machlab, Daniel Serebrenik, Daniil Sorokin, Daria Gitman, Daria Levy, Darko Stosic, David Mosallanezhad, David Yu, Davit Karamyan, Deena Donia, Deep Debroy, Deepak Narayanan, Devin O'Kelly, Dheeraj Peri, Dhruv Nathawani, Di, Wu, Dima Rekesh, Divyanshu Kakwani, Donald Plummer, Dong Anh, Dongfeng Yu, Dongfu Jiang, Donnie Kim, Dorrin Poorkay, Duncan Riach, Dusan Stosic, Dustin VanStee, Eavan Meng, Edgar Minasyan, Edward Lin, Eileen Margaret Peters Long, Elad Sarafin, Elad Segal, Elena Lantz, Ellie Evans, Elliott Ning, Eric Chung, Eric Harper, Eric Pham-Hung, Eric Tramel, Eric Yang, Erick Galinkin, Erik Pounds, Erika Goncalves Goncalves, Evan Briones, Evan Wu, Evelina Bakhturina, Evgeny Tsykunov, Ewa Dobrowolska, Faisal Ladhak, Farzan Memarian, Fay Wang, Fei Jia, Felipe Soares, Felipe Vieira Frujeri, Feng Chen, Fengguang Lin, Ferenc Galko, Frank Sun, Frankie Siino, Frida Hou, Gal Hubara Agam, Gal Kaplun, Gantavya Bhatt, Gargi Prasad, Garvit Kulshreshtha, George Armstrong, Gerald Shen, Giulio Borghesi, Gordana Neskovic, Gorkem Batmaz, Grace Lam, Greg Mason, Greg Pauloski, Grigor Nalbandyan, Grzegorz Chlebus, Grzegorz Karch, Guan-Ting Liu, Guoming Zhang, Guyue Huang, Haggai Maron, Haifeng Qian, Haim Elisha, Haoxing Ren, Haran Kumar Shiv Kumar, Haribhau Hud, Harris Nover, Harrison Saturley Hall, Hayate Iso, Helen Ngo, Herbert Hum, Herman Sahota, Hexin Wang, Himanshu Soni, Hovhannes Tamoyan, Hua Li, Huanhuan Chen, Hui Li, Hui Wang, Huy Nguyen, Ian Chiles, Ido Galil, Ido Shahaf, Igor Gitman, Igor Shovkun, Ilya Loshchilov, Ingo Guehring, Itamar Schen, Itay Levy, Itay Neeman, Ivan Moshkov, Izik Golan, Izzy Putterman, Jaemin Choi, Jakub Slowikowski, Jan Kautz, Jane Polak Scowcroft, Jared Casper, Jatin Mitra, Jeffrey Glick, Jenny Chen, Jesse Oliver, Jiacheng Xu, Jiafan Zhu, Jialin Song, Jian Zhang, Jiantao Jiao, Jiaqi Zeng, Jie Lou, Jim King, Jimmy Zhang, Jingquan Wang, Jinhang Choi, Jinju Chu, Joey Conway, Joey Guman, Johan Jatko, Johannes Rausch, John Kamalu, John Roberts, Johnny Greco, Johnny Mensel, Jonah Alben, Jonas Yang, Jonathan Cohen, Jonathan Raiman, Joseph Jennings, Joshua Mabry, Joshua Pierce, Joyjit Daw, Julien Veron Vialard, Junkeun Yi, Jupinder Parmar, Kajal Jain, Kan Zhu, Kari Briski, Katherine Cheung, Katherine Luna, Keith Willowhawk, Keith Wyss, Keshav Santhanam, Kevin Shih, Kezhi Kong, Khanh Nguyen, Khushi Bhardwaj, Kirthi Shankar Sivamani, Konstantinos Krommydas, Krishna C. Puvvada, Krzysztof Pawelec, Kumar Anik, Kyle Keprios, Kylie Day, Lawrence McAfee, Leo Du, Leon Derczynski, Li Ding, Linda Liu, Lingjie Wu, Lior Kadoch, Lizzie Wei, Luis Vega, Luke Robison, Lun Su, Maarten Van Segbroeck, Maciej Jakub Mikulski, Maer Rodrigues de Melo, Magda Sypula, Mahan Fathi, Makesh Narsimhan Sreedhar, Makesh Tarun Chandran, Manoj Kilaru, Maor Ashkenazi, Marc Cuevas, Marc Romeijn, Marcin Chochowski, Mark Cai, Mark Mozolewski, Markus Kliegl, Marta Stepniewska-Dziubinska, Martyna Patelka, Mattei

Machczynski, Matvei Novikov, Mauricio Ferrato, Maximilian Golub, Mehrzad Samadi, Melissa Corpuz, Mengru Wang, Mengxi Wu, Meredith Price, Meriem Boubdir, Micah Schaffer, Michael Andersch, Michael Boone, Michael Gschwind, Michael Lightstone, Michael Loh, Michal Bien, Michal Zawalski, Michelle Gill, Miguel Martinez, Mikail Khona, Mike Chrzanowski, Mike Houston, Mingyuan Ma, Minseok Lee, Mohamed Fawzy, Mohammad Dabbah, Mohammad Shoeybi, Mostofa Patwary, Nabin Mulepati, Najeeb Nabwani, Namit Dhameja, Narimane Hennouni, Natalie Hereth, Nathaniel Pinckney, Nave Algarici, Nave Assaf, Netanel Haber, Nicholas Knight, Nick Reamaroon, Nickson Quak, Nidhi Bhatia, Nikhil Desai, Nikolai Ludwig, Nima Tajbakhsh, Ning Xu, Nir Ailon, Nirmal Juluru, Nitin Nitin, Ofri Masad, Oleg Rybakov, Oleksii Hrinchuk, Oleksii Kuchaiev, Olivia Viessmann, Olivier Delalleau, Oluwatobi Olabiyi, Omer Ullman Argov, Omri Puny, Oren Tropp, Pablo Ribalta, Pallab Bhattacharya, Panos Lampropoulos, Parth Mannan, Pasha Shamis, Patrick Legresley, Paul Gibbons, Pavlo Molchanov, Pawel Morkisz, Peter Dykas, Peter Jin, Pierre-Yves Aquilanti, Pinky Xu, Piotr Januszewski, Piotr Laskiewicz, Pooya Jannaty, Prakash Gurumurthy, Pranav Prashant Thombre, Prasoon Varshney, Pritam Gundecha, Przemek Tredak, Puhui Meng, Qiyu Wan, Rabeeh Karimi Mahabadi, Rachel Oberman, Rachit Garg, Radha Sri-Tharan, Rahul Kandu, Rakshit Sanadhya, Ran El-Yaniv, Ran Zilberstein, Rasoul Shafipour, Ray Macalisang, Rayen Tian, Reka Kovacs, Renjie Pi, Rick Izzo, Rima Shahbazyan, Rishabh Garg, Rishi Puri, Rita Fernandes Neves, Ritchie Zhao, Ritika Borkar, Ritu Gala, Riyad Islam, Robert Clark, Robert Hesse, Robert Kirby, Roger Waleffe, Rohit Watve, Roi Koren, Ron Banner, Ruoxi Zhang, Russell J. Hewett, Ryan Prenger, Ryan Stewart, Ryota Egashira, Sadegh Mahdavi, Saee Paliwal, Sagar Singh, Sahil Modi, Salika Dave, Samantha Shinagawa, Samuel Kriman, Sandip Bhaskar, Sangkug Lym, Sanjay Kariyappa, Sanjeev Satheesh, Saran Vikas Murari, Satish Pasumarthi, Saurabh Mishra, Saurav Muralidharan, Scott Hara, Sean Narentharen, Selvaraj Anandaraj, Seonjin Na, Seonmeyong Bak, Seonmyeong Bak, Sepehr Sameni, Seph Mard, Serge Panev, Seth Henneman, Seth Poulos, Shahar Mor, Shantanu Acharya, Shaona Ghosh, Sharath Turuvekere Sreenivas, Sharon Mendelson, Shaun Kotek, Shawn Wang, Shay Aharon, Shaya Gharghabi, Sheng-Chieh Lin, Shi Chen, Shiqing Fan, Shirish Baskaran, Shreya Gopa, Shrimai Prabhumoye, Shubham Pachori, Shubham Toshniwal, Shuoyang Ding, Shwetha Krishnamurthy, Siddharth Singh, Simeng Sun, Sirshak Das, Sivakumar Arayandi Thottakara, Smita Ithape, Somshubra Majumdar, Soumye Singhal, Sri Harsha Singudasu, Sridhar Bhuvanapalli, Srimukh Veccham, Stas Sergienko, Stefania Alborghetti, Stephen Ge, Su Rong, Sugam Dipak Devare, Sukrit Rao, Sumeet Kumar Barua, Sungsoo Ha, Sunny Gai, Suriya Gunasekar, Suseella Panguluri, Suyog Gupta, Sviataslau Hinzburh, Sweta Priyadarshi, Syeda Nahida Akter, Talor Abramovich, Tan Bui, Tanay Varshney, Tatevik Ter-Hovhannisyan, Teodor-Dumitru Ene, Terry Kong, Thanh Do, Tianhe Zhang, Tiffany Moore, Tijmen Blankevoort, Tim Moon, Tiyasa Mitra, Tom Balough, Tomasz Grzegorzek, Tomasz Hliwiak, Tomer Asida, Tomer Bar Natan, Tomer Keren, Tomer Ronen, Tony Salim, Tony Wang, Traian Rebedea, Tugrul Konuk, Twinkle Vashishth, Udi Karpas, Ushnish De, Vahid Noorozi, Venkat Srinivasan, Venmugil Elango, Vibhor Agrawal, Victor Cui, Vijay Korthikanti, Vikas Mehta, Vinay Rao, Virginia Wu, Vitaly Kurin, Vitaly Lavrukhin, Vladimir Anisimov, Vu Pham, Wanli Jiang, Wasi Uddin Ahmad, Wataru Ishihara, Wei Du, Wei Ping, Weiheng Chai, Wenliang Dai, Wesley Helmholz, Will Jennings, Will Zhu, Wojciech Prazuch, Xiaowei Ren, Xiwen Yu, Yan Breek, Yang Chen, Yang Yu, Yangyi Chen, Yaniv Galron, Yashaswi Karnati, Yejin Choi, Yev Meyer, Yi-Fu Wu, Yian Zhang, Ying Lin, Yonatan Geifman, Yonggan Fu, Youngeun Kwon, Yu Yao, Yugi Guvvla, Yuki Huang, Yunsheng Liu, Zach Moshe, Zachary Newell, Zhilin Wang, Zhiyu Li, Zhongbo Zhu, Zhuolin Yang, Zihan Liu, Zijie Yan, and Zsolt-Alon Wertheimer. Nemotron 3 Ultra: Open, Efficient Mixture-of-Experts Hybrid Mamba-Transformer Model for Agentic Reasoning, June 2026. URL http://arxiv.org/abs/2606.15007. arXiv:2606.15007 [cs.CL].

OpenAI. Introducing SWE-bench Verified, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

OpenAI. Introducing GPT-5.5, September 2026a. URL https://openai.com/index/introducing-gpt-5-5/.

OpenAI. Non-interactive mode, 2026b. URL https://learn.chatgpt.com/docs/non-interactive-mode.

Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering, pp. 906–918, Athens Greece, August 2021. ACM. ISBN 978-1-4503-8562-6. doi: 10.1145/3468264.3468623. URL https://dl.acm.org/doi/10.1145/3468264.3468623.

QwenTeam. Qwen3.7: The Agent Frontier, May 2026. URL https://qwen.ai/blog?id=qwen3.7.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents, April 2025. URL http://arxiv.org/abs/2504.08703. arXiv:2504.08703 [cs.SE].

Cedric Richter and Heike Wehrheim. Learning Realistic Mutations: Bug Creation for Neural Bug Detectors. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 162–173, April 2022. doi: 10.1109/ICST53961.2022.00027. URL https://ieeexplore.ieee.org/document/9787824. ISSN: 2159-4848.

Artur Romazanov, Gordon Fraser, and Steffen Herbold. CDBench: Benchmarking the mutation testing capabilities of LLMs with code defenders. *Empirical Software Engineering*, 31(6):166, June 2026. ISSN 1573-7616. doi: 10.1007/s10664-026-10901-8. URL https://doi.org/10.1007/s10664-026-10901-8.

Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 224–234, Lake Buena Vista FL USA, October 2018. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236084. URL https://dl.acm.org/doi/10.1145/3236024.3236084.

Seonil Son, Ju-Min Oh, Heegon Jin, Cheolhun Jang, Jeongbeom Jeong, and KunTae Kim. Arenalite: Efficient and Reliable Large Language Model Evaluation via Tournament-Based Direct Comparisons. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 7057–7075, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.360. URL https://aclanthology.org/2025.emnlp-main.360/.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. BugPilot: Complex Bug Generation for Efficient Learning of SWE Skills, October 2025. URL http://arxiv.org/abs/2510.19898. arXiv:2510.19898 [cs.SE].

Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. Automated Benchmark Generation for Repository-Level Coding Tasks, March 2025. URL http://arxiv.org/abs/2503.07701. arXiv:2503.07701 [cs.SE].

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution, December 2025. URL http://arxiv.org/abs/2502.18449. arXiv:2502.18449 [cs.SE].

Wendong Xu, Jing Xiong, Chenyang Zhao, Qiujiang Chen, Haoran Wang, Hui Shen, Zhongwei Wan, Jianbo Dai, Taiqiang Wu, He Xiao, Chaofan Tao, Z. Morley Mao, Ying Sheng, Zhijiang Guo, Hongxia Yang, Bei Yu, Lingpeng Kong, Quanquan Gu, and Ngai Wong. SwingArena: Competitive Programming Arena for Long-context GitHub Issue Solving, March 2026. URL http://arxiv.org/abs/2505.23932. arXiv:2505.23932 [cs.CL].

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, November 2024. URL http://arxiv.org/abs/2405.15793. arXiv:2405.15793 [cs.SE].

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling Data for Software Engineering Agents, May 2025. URL http://arxiv.org/abs/2504.21798. arXiv:2504.21798 [cs.SE].

Z.ai. GLM-5.2: Built for Long-Horizon Tasks, June 2026. URL https://z.ai/blog/glm-5.2.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench Goes Live!, June 2025. URL http://arxiv.org/abs/2505.23419. arXiv:2505.23419 [cs.SE].

Zenong Zhang and Zach Patterson. FixReveRteR: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. 2022.

Ruochen Zhao, Wenxuan Zhang, Yew Ken Chia, Weiwen Xu, Deli Zhao, and Lidong Bing. Auto-Arena: Automating LLM Evaluations with Agent Peer Battles and Committee Discussions. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4440–4463, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.223. URL https://aclanthology.org/2025.acl-long.223/.

AI USE STATEMENT

In this work, we used generative AI to generate tasks, and attempt solutions to the tasks for evaluation of AI coding agents. We have not used generative AI tools to conceptualize our framework, propose or refine hypotheses, design research methodology or experiments, implement methods, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results. Using generative AI tools for formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, or assisting with translation are not applicable to this work. We used generative AI tools for create and polish figures or images, suggest experimental parameters, create and edit software code, and edit and format the paper to improve readability. AI-generated code was manually verified in pair programming sessions by the first and second authors, and was further verified using unit and integration test cases. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

SWE-DUEL is deliberately capable of generating subtle defects in plausible changes to real open-source code, selecting defects that evade the repository's existing CI and admitting them only when a frontier reviewer can nevertheless repair them under blind review. It is therefore an automated generator of CI-evading defects, not merely a conventional code-generation benchmark, and its prompts, trajectories, sealed tests, and admitted challenge bank are dual-use artifacts. The observed 27.3% attack success rate of the strongest Red participant against frontier-model reviewers is consequently a disclosure-relevant security finding: it demonstrates that apparently plausible, test-passing contributions can retain defects despite automated review, rather than establishing an exploit rate for deployed software.

All generation and evaluation in this study ran in isolated local containers at pinned repository revisions. We did not target deployed services, contact maintainers under false pretenses, open pull requests, or submit any generated feature, defect, test, or patch to a real repository. If future use of the pipeline reveals an independently existing vulnerability rather than an introduced benchmark defect, we require coordinated disclosure to the affected maintainer before publication. These safeguards do not eliminate the dual-use risk, and we will revisit access decisions if evidence suggests that release would materially increase practical misuse.

REPRODUCIBILITY STATEMENT

We have open-sourced the SWE-DUEL toolkit (Anonymous, 2026b) and the experiment dataset (Anonymous, 2026a) for reproducibility.

A TASK FRESHNESS AND TASK BANK RELEASE POLICY

The same task cannot remain both public and unseen. In SWE-DUEL, a generated bank is static within a tournament so every opponent faces controlled, reusable tasks; freshness is renewed across tournaments by generating a new task bank. Full publication of replicated results would enable exact external replay, but it would both make the tasks unsuitable for future unseen-task claims. We encourage the generating of new task bank for each replication study to limit the possibility of the released tasks from previous replication studies leaking into the training data of newer LLMs. Any disclosed task bank should be marked *retired* and cannot support evaluation of models released in the future, and other unseen-task-related works.

B COMPLETE WORKED TASKS

B.1 PYTHON: SQLALCHEMY LRUCACHE.GET_OR_CREATE

Artifacts. Task `e1a27fac-1285-4690-ab55-cb8fabca312d`; successful solving run `cd538560-b183-4eba-a59f-dc7862e90ba8`.

Feature request. Add `LRUCache.get_or_create(key, creator)`, which returns and refreshes an existing cached value on a hit, or calls a zero-argument creator on a miss, stores the created value through normal LRU size management, and returns it. If the creator raises, no entry is stored; if it reentrantly populates the key, that value is returned instead of being overwritten.

Complete PR diff.

```
--- a/lib/sqlalchemy/util/_collections.py
+++ b/lib/sqlalchemy/util/_collections.py
@@ -506,6 +506,29 @@
506          else:
507              return default
509 +    def get_or_create(self, key: _KT, creator: Callable[[], _VT]) -> _VT:
510 +        """Return a value from the cache, creating it if absent.
511 +
512 +        This provides an atomic-looking convenience for the common cache
513 +        pattern of retrieving a value and lazily populating it on a miss,
514 +        while preserving normal LRU access tracking for hits.
515 +        """
516 +
517 +        value = self.get(key)
518 +        if value is not None:
519 +            return value
520 +
521 +        value = creator()
522 +
523 +        item = self._data.get(key)
524 +        if item is not None:
525 +            item[2][0] = self._inc_counter()
526 +            return item[1]
527 +
528 +        self._data[key] = (key, value, [self._inc_counter()])
529 +        self._manage_size()
530 +        return value
531 +
532      def __getitem__(self, key: _KT) -> _VT:
533          item = self._data[key]
534          item[2][0] = self._inc_counter()
```

Embedded defect (line 517). The initial `get(key)` uses `None` as the implicit default, so a cached `None` is treated as a miss. A side-effecting creator can run and valid cached state can be overwritten or evicted.

Complete sealed bug test.

```python
from sqlalchemy import util


def test_lru_get_or_create_treats_cached_none_as_existing_value():
    cache = util.LRUCache(2, threshold=0)
    calls = []

    cache["none-key"] = None
    cache["other-key"] = "other-value"

    value = cache.get_or_create(
        "none-key", lambda: calls.append("created") or "new-value"
    )
    cache["new-key"] = "new-entry"

    assert value is None
    assert calls == []
    assert "none-key" in cache
    assert "other-key" not in cache
    assert "new-key" in cache
```

Complete successful Blue patch.

```
--- a/lib/sqlalchemy/util/_collections.py
+++ b/lib/sqlalchemy/util/_collections.py
@@ -506,6 +506,28 @@
506          else:
```

```
            return default

+    def get_or_create(self, key: _KT, creator: Callable[[], _VT]) -> _VT:
+        """Return a cached value, creating and caching it if absent.
+
+        If the creator populates *key* reentrantly, that value takes
+        precedence over the value returned by the creator.
+        """
+        item = self._data.get(key)
+        if item is not None:
+            item[2][0] = self._inc_counter()
+            return item[1]
+
+        value = creator()
+
+        item = self._data.get(key)
+        if item is not None:
+            item[2][0] = self._inc_counter()
+            return item[1]
+
+        self._data[key] = (key, value, [self._inc_counter()])
+        self._manage_size()
+        return value
+
     def __getitem__(self, key: _KT) -> _VT:
         item = self._data[key]
         item[2][0] = self._inc_counter()
```

B.2 GO: CHI MUX.ALLOWEDMETHODS

Artifacts. Task `af3fb594-76dd-4f6e-8570-42130eb346d3`; successful solving run `dff4a747-68d3-4e68-9850-5f54b288432e`.

Feature request. Add `(*Mux).AllowedMethods(path string) []string`, returning a stable, sorted list of methods registered for a concrete path. Lookup normalizes an empty path to `/`, supports parameters, mounted subrouters, catch-alls, and custom methods, and returns `nil` when no route matches.

Complete PR diff.

```
--- a/mux.go
+++ b/mux.go
@@ -4,6 +4,7 @@
  "context"
  "fmt"
  "net/http"
+ "sort"
  "strings"
  "sync"
 )
@@ -360,6 +361,41 @@
  return mx.Find(rctx, method, path) != ""
 }

+// AllowedMethods returns the HTTP methods with a registered handler for path.
+//
+// The path is matched the same way as an incoming request path, including
+// parameters, catch-all routes, mounted subrouters, and custom methods
+// registered with RegisterMethod. The returned methods are sorted for stable
+// introspection output. If no route matches path, it returns nil.
+func (mx *Mux) AllowedMethods(path string) []string {
+ if path == "" {
+  path = "/"
+ }
+
+ methods := make([]string, 0, len(methodMap))
+ for method := range methodMap {
+  methods = append(methods, method)
+ }
+ sort.Strings(methods)
+
+ rctx := NewRouteContext()
+ allowed := methods[:0]
+ for _, method := range methods {
+  rctx.Reset()
```

```
+   if mx.Find(rctx, method, path) != "" {
+     allowed = append(allowed, method)
+     if len(rctx.RoutePatterns) > 2 {
+       break
+     }
+   }
+ }
+ if len(allowed) == 0 {
+   return nil
+ }
+
+ return append([]string(nil), allowed...)
+}
+
 // Find searches the routing tree for the pattern that matches
 // the method/path.
 //
```

Embedded defect (lines 387–389). After the first method match, lookup stops whenever nested mounting produced more than two route patterns. Other methods registered at the same nested path are omitted, which can produce an incomplete `Allow` or CORS response.

Complete sealed bug test.

```go
package acaatest

import (
  "net/http"
  "reflect"
  "testing"

  "github.com/go-chi/chi/v5"
)

func TestAllowedMethodsAcrossNestedMountedSubrouters(t *testing.T) {
  leaf := chi.NewRouter()
  leaf.Delete("/items/{id}", noopBugHandler)
  leaf.Patch("/items/{id}", noopBugHandler)

  versioned := chi.NewRouter()
  versioned.Mount("/v1", leaf)

  r := chi.NewRouter()
  r.Mount("/api", versioned)

  got := r.AllowedMethods("/api/v1/items/abc123")
  want := []string{http.MethodDelete, http.MethodPatch}
  if !reflect.DeepEqual(got, want) {
    t.Fatalf("AllowedMethods returned %v, want %v", got, want)
  }
}

func noopBugHandler(w http.ResponseWriter, r *http.Request) {}
```

Complete successful Blue patch.

```
--- a/mux.go
+++ b/mux.go
@@ -4,6 +4,7 @@
  "context"
  "fmt"
  "net/http"
+ "sort"
  "strings"
  "sync"
 )
@@ -360,6 +361,38 @@
  return mx.Find(rctx, method, path) != ""
 }

+// AllowedMethods returns the HTTP methods with a registered handler for path.
+//
+// The path is matched the same way as an incoming request path, including
+// parameters, catch-all routes, mounted subrouters, and custom methods
+// registered with RegisterMethod. The returned methods are sorted for stable
```

```
+// introspection output. If no route matches path, it returns nil.
+func (mx *Mux) AllowedMethods(path string) []string {
+ if path == "" {
+   path = "/"
+ }
+
+ methods := make([]string, 0, len(methodMap))
+ for method := range methodMap {
+   methods = append(methods, method)
+ }
+ sort.Strings(methods)
+
+ rctx := NewRouteContext()
+ allowed := methods[:0]
+ for _, method := range methods {
+   rctx.Reset()
+   if mx.Find(rctx, method, path) != "" {
+     allowed = append(allowed, method)
+   }
+ }
+ if len(allowed) == 0 {
+   return nil
+ }
+
+ return append([]string(nil), allowed...)
+}
+
 // Find searches the routing tree for the pattern that matches
 // the method/path.
 //
```

B.3 NODE: NODE-JSONWEBTOKEN REQUIREDCLAIMS

Artifacts. Task `fb6f0e8c-b6e9-4bd5-a2eb-4970d8856ae1`; successful solving run `f1b3beb2-769c-438a-a9a1-bf656a5d45d7`.

Feature request. Add a `requiredClaims` option to `jwt.verify` accepting one claim name or an array. After signature and registered-claim checks, every listed claim must be present (not `undefined`) or verification fails with `JsonWebTokenError`; the option must be a nonempty string or nonempty array of nonempty strings.

Complete PR diff.

```
--- a/verify.js
+++ b/verify.js
@@ -54,6 +54,13 @@
     return done(new JsonWebTokenError('allowInvalidAsymmetricKeyTypes must be a boolean')
   ↪ );
   }

+  if (options.requiredClaims !== undefined) {
+    const requiredClaims = Array.isArray(options.requiredClaims) ? options.requiredClaims
   ↪  : [options.requiredClaims];
+    if (requiredClaims.length === 0 || !requiredClaims.every(function (claim) { return
   ↪ typeof claim === 'string' && claim.trim() !== ''; })) {
+      return done(new JsonWebTokenError('requiredClaims must be a non-empty string or an
   ↪ array of non-empty strings'));
+    }
+  }
+
   const clockTimestamp = options.clockTimestamp || Math.floor(Date.now() / 1000);

   if (!jwtString){
@@ -234,6 +241,7 @@
       }
     }

+
     if (options.maxAge) {
       if (typeof payload.iat !== 'number') {
         return done(new JsonWebTokenError('iat required when maxAge is specified'));
@@ -258,6 +266,16 @@
       });
     }
```

```
269 +    if (options.requiredClaims) {
270 +      const requiredClaims = Array.isArray(options.requiredClaims) ? options.
    ↪ requiredClaims : [options.requiredClaims];
271 +      for (let i = 0; i < requiredClaims.length; i++) {
272 +        const claim = requiredClaims[i];
273 +        if (typeof payload[claim] === 'undefined') {
274 +          return done(new JsonWebTokenError('jwt claim "' + claim + '" is missing'));
275 +        }
276 +      }
277 +    }
278 +
279      return done(null, payload);
280    });
281  };
```

Embedded defect (lines 269–277). With `complete: true`, verification returns the complete token before reaching the new check. Mandatory authorization claims can therefore be absent without error.

Complete sealed bug test.

```
1  'use strict';

3  const assert = require('assert');
4  const jwt = require('../');

6  describe('requiredClaims with complete option', function () {
7    const secret = 'shhhhh';

9    it('should still reject a token missing a required claim when complete is true',
     ↪ function (done) {
10     const token = jwt.sign({ role: 'admin' }, secret, { algorithm: 'HS256' });
11     jwt.verify(token, secret, { requiredClaims: 'permission', complete: true }, function (
     ↪ err, decoded) {
12       assert.ok(err, 'expected an error when a required claim is missing');
13       assert.strictEqual(err.name, 'JsonWebTokenError');
14       assert.ok(err.message.includes('jwt claim "permission" is missing'));
15       assert.strictEqual(decoded, undefined);
16       done();
17     });
18   });
19 });
```

Complete successful Blue patch.

```
   --- a/verify.js
   +++ b/verify.js
   @@ -54,6 +54,13 @@
54       return done(new JsonWebTokenError('allowInvalidAsymmetricKeyTypes must be a boolean')
     ↪ );
55     }

57 +   if (options.requiredClaims !== undefined) {
58 +     const requiredClaims = Array.isArray(options.requiredClaims) ? options.requiredClaims
     ↪  : [options.requiredClaims];
59 +     if (requiredClaims.length === 0 || !requiredClaims.every(function (claim) { return
     ↪ typeof claim === 'string' && claim.trim() !== ''; })) {
60 +       return done(new JsonWebTokenError('requiredClaims must be a non-empty string or an
     ↪ array of non-empty strings'));
61 +     }
62 +   }
63 +
64     const clockTimestamp = options.clockTimestamp || Math.floor(Date.now() / 1000);

66     if (!jwtString){
   @@ -248,6 +255,16 @@
255        }
256      }

258 +    if (options.requiredClaims) {
259 +      const requiredClaims = Array.isArray(options.requiredClaims) ? options.
     ↪ requiredClaims : [options.requiredClaims];
260 +      for (let i = 0; i < requiredClaims.length; i++) {
261 +        const claim = requiredClaims[i];
262 +        if (typeof payload[claim] === 'undefined') {
```

```
+          return done(new JsonWebTokenError('jwt claim "' + claim + '" is missing'));
+        }
+      }
+    }
+
     if (options.complete === true) {
       const signature = decodedToken.signature;
```

B.4 JAVA: OWASP SANITIZER DESCRIPTION LISTS

Artifacts. Task `eef5eb63-11e9-4e2f-9810-1bc40816e6a2`; successful solving run `369e5c91-9b26-48c4-8cf5-eebe3cce83f3`.

Feature request. Add a pre-packaged `Sanitizers.DESCRIPTION_LISTS` policy allowing `dl`, `dt`, and `dd` without attributes, composable with policies such as `FORMATTING` or `BLOCKS`.

Complete PR diff.

```
--- a/owasp-java-html-sanitizer/src/main/java/org/owasp/html/Sanitizers.java
+++ b/owasp-java-html-sanitizer/src/main/java/org/owasp/html/Sanitizers.java
@@ -96,6 +96,22 @@
     .allowTextIn("table")  // WIDGY
     .toFactory();

+  /**
+   * Allows description list elements: {@code <dl>}, {@code <dt>}, and
+   * {@code <dd>}.  These are useful for glossaries, metadata displays,
+   * and FAQ sections.  Does not allow any attributes on these elements.
+   * <p>
+   * Combine with {@link #FORMATTING} to allow inline formatting inside
+   * {@code <dt>} and {@code <dd>} elements, e.g.
+   * <pre>
+   *   PolicyFactory sanitizer = Sanitizers.DESCRIPTION_LISTS
+   *       .and(Sanitizers.FORMATTING);
+   * </pre>
+   */
+  public static final PolicyFactory DESCRIPTION_LISTS = new HtmlPolicyBuilder()
+      .allowElements("dl", "dt", "dd", "td")
+      .toFactory();
+
   private static final AttributePolicy INTEGER = new AttributePolicy() {
     public String apply(
         String elementName, String attributeName, String value) {
```

Embedded defect (line 112). The allowlist accidentally includes `td`. This visually similar copy-paste token preserves table cells despite the policy's documented description-list-only contract.

Complete sealed bug test.

```
import static org.junit.Assert.*;

import org.junit.Test;
import org.owasp.html.Sanitizers;

@SuppressWarnings("javadoc")
public class TestAcaaBug {

  @Test
  public void testDescriptionListsDoesNotAllowTableDataCell() {
    String input = "<td>Should be stripped</td>";
    String want = "Should be stripped";
    String got = Sanitizers.DESCRIPTION_LISTS.sanitize(input);
    assertEquals(want, got);
  }

  @Test
  public void testDescriptionListsStripsTableElementsInsideDl() {
    String input = "<dl><dt>Term</dt><dd>Definition</dd><td>sneaky cell</td></dl>";
    String want = "<dl><dt>Term</dt><dd>Definition</dd>sneaky cell</dl>";
    String got = Sanitizers.DESCRIPTION_LISTS.sanitize(input);
    assertEquals(want, got);
  }
```

```
  @Test
  public void testDescriptionListsDoesNotAllowTableHeaderCell() {
    String input = "<th>Table header</th>";
    String want = "Table header";
    String got = Sanitizers.DESCRIPTION_LISTS.sanitize(input);
    assertEquals(want, got);
  }
}
```

Complete successful Blue patch.

```
--- a/owasp-java-html-sanitizer/src/main/java/org/owasp/html/Sanitizers.java
+++ b/owasp-java-html-sanitizer/src/main/java/org/owasp/html/Sanitizers.java
@@ -96,6 +96,22 @@
     .allowTextIn("table")  // WIDGY
     .toFactory();

+  /**
+   * Allows description list elements: {@code <dl>}, {@code <dt>}, and
+   * {@code <dd>}.  These are useful for glossaries, metadata displays,
+   * and FAQ sections.  Does not allow any attributes on these elements.
+   * <p>
+   * Combine with {@link #FORMATTING} to allow inline formatting inside
+   * {@code <dt>} and {@code <dd>} elements, e.g.
+   * <pre>
+   *   PolicyFactory sanitizer = Sanitizers.DESCRIPTION_LISTS
+   *       .and(Sanitizers.FORMATTING);
+   * </pre>
+   */
+  public static final PolicyFactory DESCRIPTION_LISTS = new HtmlPolicyBuilder()
+      .allowElements("dl", "dt", "dd")
+      .toFactory();
+
   private static final AttributePolicy INTEGER = new AttributePolicy() {
     public String apply(
         String elementName, String attributeName, String value) {
```

B.5 C/C++: cJSON BOOLEAN ARRAYS

Artifacts. Task `b39a0522-ce05-4a16-8486-56425000ef52`; successful solving run `aadee125-8b6a-4c3a-9ccb-7935b025dd48`.

Feature request. Add `cJSON_CreateBoolArray(const cJSON_bool *booleans, int count)`, returning a cJSON array whose boolean items correspond to the input values.

Complete PR diff.

```
--- a/cJSON.c
+++ b/cJSON.c
@@ -2603,6 +2603,46 @@
 }

 /* Create Arrays: */
+CJSON_PUBLIC(cJSON *) cJSON_CreateBoolArray(const cJSON_bool *booleans, int count)
+{
+    size_t i = 0;
+    cJSON *n = NULL;
+    cJSON *p = NULL;
+    cJSON *a = NULL;
+
+    if ((count < 0) || (booleans == NULL))
+    {
+        return NULL;
+    }
+
+    a = cJSON_CreateArray();
+
+    for(i = 0; a && (i < (size_t)count); i++)
+    {
+        n = cJSON_CreateBool(booleans[i]);
+        if (!n)
+        {
```

```
+            cJSON_Delete(a);
+            return NULL;
+        }
+        if(!i)
+        {
+            a->child = n;
+        }
+        else
+        {
+            suffix_object(p, n);
+        }
+        p = n;
+    }
+
+    if (a && a->child) {
+        a->child->prev = a->child;
+    }
+
+    return a;
+}
+
 CJSON_PUBLIC(cJSON *) cJSON_CreateIntArray(const int *numbers, int count)
 {
     size_t i = 0;
--- a/cJSON.h
+++ b/cJSON.h
@@ -219,6 +219,7 @@

 /* These utilities create an Array of count items.
  * The parameter count cannot be greater than the number of elements in the number array, otherwise array access will be out of bounds.*/
+CJSON_PUBLIC(cJSON *) cJSON_CreateBoolArray(const cJSON_bool *booleans, int count);
 CJSON_PUBLIC(cJSON *) cJSON_CreateIntArray(const int *numbers, int count);
 CJSON_PUBLIC(cJSON *) cJSON_CreateFloatArray(const float *numbers, int count);
 CJSON_PUBLIC(cJSON *) cJSON_CreateDoubleArray(const double *numbers, int count);
```

Embedded defect (lines 2639–2641). The array's tail pointer, `a->child->prev`, points to the first element rather than the last element `n`. Subsequent constant-time append uses the wrong tail and can corrupt list order or structure.

Complete sealed bug test.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "cJSON.h"

void test_bug(void) {
    cJSON_bool booleans[3] = {1, 0, 1};
    cJSON *array = cJSON_CreateBoolArray(booleans, 3);

    assert(array != NULL);
    assert(cJSON_IsArray(array));
    assert(cJSON_GetArraySize(array) == 3);

    assert(array->child != NULL);
    assert(array->child->prev != NULL);
    assert(array->child->prev == cJSON_GetArrayItem(array, 2));

    cJSON *new_item = cJSON_CreateBool(1);
    cJSON_AddItemToArray(array, new_item);

    assert(cJSON_GetArraySize(array) == 4);
    assert(cJSON_GetArrayItem(array, 3) == new_item);
    assert(array->child->prev == new_item);

    cJSON_Delete(array);
}

int main(void) {
    test_bug();
    printf("All tests passed!\n");
    return 0;
}
```

Complete successful Blue patch.

```
--- a/cJSON.c
+++ b/cJSON.c
@@ -2602,6 +2602,46 @@
     return item;
 }

+CJSON_PUBLIC(cJSON *) cJSON_CreateBoolArray(const cJSON_bool *booleans, int count)
+{
+    size_t i = 0;
+    cJSON *n = NULL;
+    cJSON *p = NULL;
+    cJSON *a = NULL;
+
+    if ((count < 0) || (booleans == NULL))
+    {
+        return NULL;
+    }
+
+    a = cJSON_CreateArray();
+
+    for(i = 0; a && (i < (size_t)count); i++)
+    {
+        n = cJSON_CreateBool(booleans[i]);
+        if (!n)
+        {
+            cJSON_Delete(a);
+            return NULL;
+        }
+        if(!i)
+        {
+            a->child = n;
+        }
+        else
+        {
+            suffix_object(p, n);
+        }
+        p = n;
+    }
+
+    if (a && a->child) {
+        a->child->prev = n;
+    }
+
+    return a;
+}
+
 /* Create Arrays: */
 CJSON_PUBLIC(cJSON *) cJSON_CreateIntArray(const int *numbers, int count)
 {
--- a/cJSON.h
+++ b/cJSON.h
@@ -219,6 +219,7 @@

 /* These utilities create an Array of count items.
  * The parameter count cannot be greater than the number of elements in the number array, otherwise array access will be out of bounds.*/
+CJSON_PUBLIC(cJSON *) cJSON_CreateBoolArray(const cJSON_bool *booleans, int count);
 CJSON_PUBLIC(cJSON *) cJSON_CreateIntArray(const int *numbers, int count);
 CJSON_PUBLIC(cJSON *) cJSON_CreateFloatArray(const float *numbers, int count);
 CJSON_PUBLIC(cJSON *) cJSON_CreateDoubleArray(const double *numbers, int count);
```

C REPOSITORY FIELD

Table C shows the 15 repositories across 5 languages used in SWE-DUEL along with exact commit number for each of the repositories.

Table 4: The fixed repository field. Commit is the 12-hex prefix of the checked-out SHA. SLoC gives a rough estimate on the size/complexity of the repository via the number of nonblank lines in Git-tracked files of the repository's implementation language, including tests but excluding documentation and build outputs. Tests is the number of test cases in existing-suite; † denotes configured executable targets when the runner does not expose case totals. Time is the median wall-clock time of those existing-suite runs; commands that compile include compilation time.

Repository	Commit	Language	SLoC	Tests	Time (s)
Flask	`7fff56f5172c`	Python	13,751	480	3.7
Jinja	`15206881c006`	Python	18,903	909	4.2
SQLAlchemy	`5237d923ddb0`	Python	511,172	18,020	161.8
chi	`3b171578ca44`	Go	9,923	254	36.1
golang-jwt/jwt	`7ceae619e739`	Go	5,503	201	5.6
gorilla/csrf	`9dd6af1f6d30`	Go	2,760	50	3.9
Helmet	`638e43becc81`	Node	3,373	115	10.0
Express	`dbac741a49a5`	Node	17,339	1,130	6.3
node-jsonwebtoken	`ed59e76ea37a`	Node	3,949	512	3.4
OWASP Java HTML Sanitizer	`4e186b8a8c84`	Java	22,492	332	82.4
java-jwt	`695fd2bea64b`	Java	11,976	1†	81.5
JJWT	`0df975627c2c`	Java	66,626	20	127.6
cJSON	`c859b25da029`	C	18,999	1†	0.7
libexpat	`c61098da494e`	C	36,207	1†	22.6
simdjson	`1bcf71bd8505`	C++	313,998	4†	59.0

D Additional Experimental Detail

D.1 Harness Setup

All participants receives an initial step budget of 50 harness steps. The 50-step initial ceiling applies separately to Red feature generation, Red bug embedding, blind self-review, and each Blue task-solving sessions. Each of the sessions are also subject to a 30-minute wall-clock limit. If required artifacts remain missing, up to three recovery turns of six steps each is given to the participant. In this work, we choose mini-swe-agent (Yang et al., 2024), Codex CLI (OpenAI, 2026b), and Claude Code (Anthropic, 2026b), all of which are popular harnesses that post strong performances across various coding benchmarks. Note that different models use different amount of compute resources, especially depending on the accompanying harness. Thus, the harness step counts do not represent equal compute. Appendix R reports a token-level compute-controlled analysis. All the agents, validation gates, and turn scorers run in separate Docker container instances of the same repository-specific image to ensure a consistent environment. The inference temperature for all the models is fixed at 0.2.

D.2 Model API Access Dates

All participants access their models through the OpenRouter API[1]. Table 5 lists the first and last recorded API access date for each model. The windows span task creation (including failed attempts and blind self-review validation), evaluation in the main and harness-ablation tournaments, and the post-hoc studies of Appendices I, J, and K: the adaptive task difficulty study adds the GLM-5.1 and GLM-5.3 windows (adaptive task creation), and extends the DeepSeek V4 Pro window (fixed-reference task solver for the adaptive tasks). They are identical across the harnesses paired with a model because all harnesses route to the same OpenRouter endpoint; all dates are in UTC.

[1] `https://openrouter.ai`

Table 5: Per-model API access dates. Each entry gives the first and last recorded model call through the OpenRouter API (UTC). All calls of a model use the same OpenRouter model identifier regardless of the paired harness.

Model	OpenRouter model ID	First access	Last access
GPT-5.5	`openai/gpt-5.5`	2026-06-18	2026-08-29
Claude Opus 4.8	`anthropic/claude-opus-4.8`	2026-06-18	2026-08-29
Kimi K2.7 Code	`moonshotai/kimi-k2.7-code`	2026-06-18	2026-08-29
DeepSeek V4 Pro	`deepseek/deepseek-v4-pro`	2026-06-18	2026-09-19
GLM 5.1	`z-ai/glm-5.1`	2026-09-18	2026-09-18
GLM 5.2	`z-ai/glm-5.2`	2026-06-18	2026-08-30
GLM 5.3	`z-ai/glm-5.3`	2026-09-18	2026-09-18
Qwen 3.7 Max	`qwen/qwen3.7-max`	2026-07-11	2026-08-29
Nemotron 3 Ultra	`nvidia/nemotron-3-ultra-550b-a55b`	2026-07-11	2026-08-29
MiniMax M3	`minimax/minimax-m3`	2026-06-18	2026-08-29
Gemini 3.1 Pro	`google/gemini-3.1-pro-preview`	2026-06-19	2026-08-29

E Evaluation and Outcome Classification

Algorithm 1 describes the end-to-end procedure encompassing task cache lookup, and turn-based match scoring. The $\mathrm{Load}(r, R_j, i)$ procedure returns the ith admitted task for Red participant r in bank-index order. After Blue submits a diff, the existing test suite, and the test suites for feature, and embedded bugs executes. Existing test suite failure classifies the attempt as a regression. If existing tests pass but the feature suite fails, the attempt is a feature collapse. If both pass but hidden bug tests fail, the defect shipped. Blue wins if and only if all the test suites pass.

F Rank Trajectories

Figure 5 re-fits BT using only the matches available at each scheduled round. It documents when observations arrived, rather than a second estimand: the reported leaderboard fits the complete persisted schedule in Table 1.

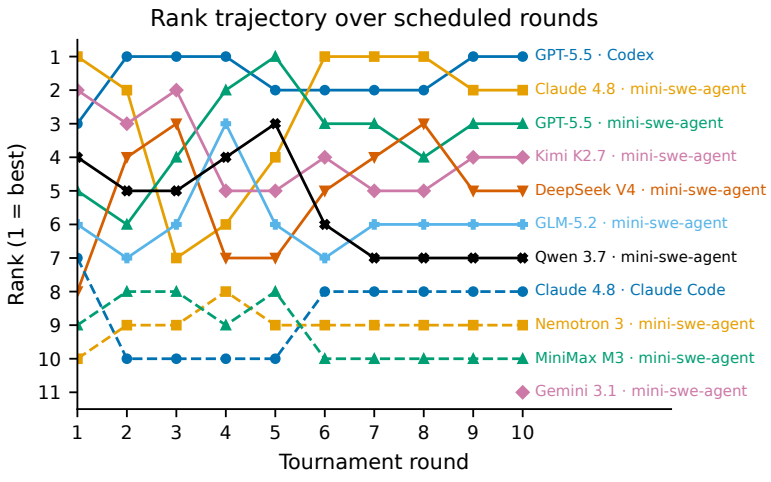


Figure 5: BT rank recomputed after each scheduled tournament round. A late entrant begins only when it first plays. This arrival-order diagnostic is not a ranking rule or convergence criterion.

Algorithm 1 Deterministic execution and scoring of a symmetric match

Require: Participants A, B; repositories $\mathcal{R}$; target count N; task bank $\mathcal{B}$; solving cache $\mathcal{D}$
Ensure: Match totals (Q_A, Q_B) and outcome O

$Q_A \leftarrow 0$; $Q_B \leftarrow 0$; $T \leftarrow [\,]$
for all $R_j \in \mathcal{R}$ **do**
  for $i \leftarrow 1$ **to** N **do**
    for all $(r, b) \in [(A, B), (B, A)]$ **do** ▷ swap Red and Blue roles
      $C \leftarrow \mathrm{Load}(\mathcal{B}, r, R_j, i)$
      if $C = \varnothing$ **then**
        $(S_{\mathrm{Blue}}, S_{\mathrm{Red}}) \leftarrow (1, 0)$ ▷ automatic Blue win for missing Red task
      else
        if $\mathcal{D}$ contains (C, b) **then**
          $F \leftarrow \mathcal{D}[C, b]$ ▷ reuse immutable solving run
        else
          $P \leftarrow \textsc{PublicBundle}(C)$
          $F \leftarrow \textsc{Solve}(b, P)$ ▷ sealed defect metadata and bug tests remain hidden
          $\mathcal{D}[C, b] \leftarrow F$
        end if
        $S_e \leftarrow \mathbb{1}[\textsc{ExistingTests}(C, F) \text{ pass}]$
        $S_f \leftarrow \mathbb{1}[\textsc{FeatureTests}(C, F) \text{ pass}]$
        $S_b \leftarrow \mathbb{1}[\textsc{HiddenBugTests}(C, F) \text{ pass}]$
        $S_{\mathrm{Blue}} \leftarrow S_e S_f S_b$; $S_{\mathrm{Red}} \leftarrow 1 - S_{\mathrm{Blue}}$
      end if
      $Q_b \leftarrow Q_b + S_{\mathrm{Blue}}$; $Q_r \leftarrow Q_r + S_{\mathrm{Red}}$
      $\textsc{Append}(T, (R_j, i, r, b, C, S_{\mathrm{Red}}, S_{\mathrm{Blue}}))$
    end for
  end for
end for
if $Q_A > Q_B$ **then**
  $O \leftarrow A$-wins
else if $Q_B > Q_A$ **then**
  $O \leftarrow B$-wins
else
  $O \leftarrow \mathrm{draw}$
end if
return (Q_A, Q_B, O, T)

G Manual Audit for Sealed-oracle False Negatives

The sealed bug test is authored by *Red* agent, for deterministic verification of the embedded bug's existence. However, it does not guarantee that it accepts every semantically valid repair. Therefore, two authors conducted a post-hoc manually audit on bug fix failures where existing and feature suites pass but the sealed bug suite fails. The main tournament contains 158 turns with such outcomes. We sorted them by turn UUID and drew 50 without replacement using Python's `random.Random(20260823).sample`. For each sampled turn, we jointly inspected the feature specification, admitted PR diff, defect description and location, sealed test, Blue explanation, and submitted patch. A *genuine miss* means *Blue* agent leaves the embedded bug unrepaired while a *false negative* means *Blue* agent implements a legitimate repair but fails because the sealed oracle requires an unnecessarily specific implementation for the bug fix.

Result. Of 50 sampled outcomes, 45 were genuine misses, three were false negatives, and two were ambiguous. Treating every sampled scored failure as the denominator, the estimated false-negative rate is $3/50 = 6.0\%$ (95% Wilson interval 2.1–16.2%). Excluding the two ambiguous cases gives $3/48 = 6.25\%$ (95% Wilson interval 2.1–16.8%). All three false negatives arose from two Go JWT tasks: Blue rejected an unverifiable custom-claims token with `ErrTokenUnverifiable` rather than the oracle's required `ErrTokenRequiredClaimMissing` (two solving runs of one task), or correctly treated future-`iat` validation as the existing `WithIssuedAt` option's respon-

sibility while the oracle required `WithMaxTokenAge` to duplicate it. Thus, over-specification is uncommon in this sample.

H Main Tournament Bradley–Terry Confidence Intervals

Table 6 shows BT intervals and central 95% rank intervals use the 1,000-draw match-bootstrap samples. No two adjacent participants in the ranking ordering is distinguishable at 95%. Such overlapping confidence intervals among nearby participants are common in head-to-head arenas (Arena.ai, 2026; designarena.ai, 2026).

Table 6: BT point estimate and ranking confidence intervals corresponding to Figure 1.

Competitor	BT 95% CI	95% rank interval
GPT-5.5 [codex]	$[+1.30, +9.48]$	1–5
Claude Opus 4.8 [mini-swe-agent]	$[+1.10, +8.95]$	1–6
GPT-5.5 [mini-swe-agent]	$[-0.57, +6.95]$	1–7
Kimi-K2.7-Code [mini-swe-agent]	$[-0.07, +7.44]$	1–7
DeepSeek-V4-Pro [mini-swe-agent]	$[-0.61, +5.87]$	1–7
GLM-5.2 [mini-swe-agent]	$[-1.00, +4.44]$	1–7
Qwen3.7-Max [mini-swe-agent]	$[-1.45, +3.34]$	4–8
Claude Opus 4.8 [claude-code]	$[-6.91, +1.33]$	7–9
Nemotron-3-Ultra [mini-swe-agent]	$[-12.57, -1.33]$	8–11
MiniMax-M3 [mini-swe-agent]	$[-8.45, -2.34]$	8–11
Gemini-3.1-Pro [mini-swe-agent]	$[-12.57, -2.45]$	9–11

I Session-C Admission Stability

Admission requires that a fresh blind instance of the same model–harness repair the embedded defect (Section 3.2). All models' inference is configured with temperature 0.2, so the reviewer's trajectory (and hence its repair) is a stochastic draw rather than a deterministic property. A task that no fresh reviewer repairs (for e.g. fails all k independent re-runs) is therefore a direct signature of a *lucky admission*. Thus, we re-run the identical Session-C probe using the same model–harness pair, prompt template, public test bundles, sealed bug-test oracles and other harness settings such as recovery loop, step and time budgets. The probe runs for $k{=}3$ additional times per task on a stratified sample of the admitted tasks the headline tournament actually solved: four tasks per participant, round-robined across repositories, drawn from the 134 tournament-contested tasks by a per-participant seeded random generator. Each of the $44 \times 3 = 132$ runs launches a fresh reviewer instance in an isolated workspace and persists its verdict, trajectory, and bug-test output.

Table 7: Session-C admission-noise stability study over the headline tournament bank. For each participant, its four sampled tournament tasks distribute over the repaired-$x/3$ buckets according to how many of the $k{=}3$ fresh blind re-runs of the exact Session-C probe repaired them; "Repairs/runs" pools the $44 \times 3 = 132$ per-run verdicts with Wilson 95% intervals.

Participant	n	3/3	2/3	1/3	0/3	Repairs/runs	Rate (95% Wilson)
DeepSeek-V4-Pro [mini-swe-agent]	4	4	–	–	–	12/12	100.0% [75.7, 100.0]
Kimi-K2.7-Code [mini-swe-agent]	4	4	–	–	–	12/12	100.0% [75.7, 100.0]
GPT-5.5 [Codex CLI]	4	4	–	–	–	12/12	100.0% [75.7, 100.0]
GPT-5.5 [mini-swe-agent]	4	4	–	–	–	12/12	100.0% [75.7, 100.0]
Claude Opus 4.8 [Claude Code]	4	3	1	–	–	11/12	91.7% [64.6, 98.5]
Gemini-3.1-Pro [mini-swe-agent]	4	3	1	–	–	11/12	91.7% [64.6, 98.5]
Nemotron-3-Ultra [mini-swe-agent]	4	3	1	–	–	11/12	91.7% [64.6, 98.5]
Qwen3.7-Max [mini-swe-agent]	4	3	1	–	–	11/12	91.7% [64.6, 98.5]
MiniMax-M3 [mini-swe-agent]	4	2	2	–	–	10/12	83.3% [55.2, 95.3]
GLM-5.2 [mini-swe-agent]	4	2	2	–	–	10/12	83.3% [55.2, 95.3]
Claude Opus 4.8 [mini-swe-agent]	4	3	–	–	1	9/12	75.0% [46.8, 91.1]
All participants	**44**	**35**	**8**	**0**	**1**	**121/132**	**91.7%** [85.7, 95.3]

Result. The distribution demonstrates the admitted tasks' reliable solvability: 35 of 44 tasks (79.5%) are repaired 3/3, 8 (18.2%) are repaired 2/3, none 1/3, and a single task (2.3%) is repaired 0/3—the study's one lucky admission. The pooled per-run repair rate is $121/132 = 91.7\%$ (Wilson 95% CI $[85.7\%, 95.3\%]$), and the average per-run repair probability over the tournament bank is approximately 0.92. Thus, admission mostly certifies reliable solvability rather than a lucky draw. Table 7 and Figure 6 report the per-participant breakdown. "Repaired $x/3$" counts how many of the fresh blind reviewers repaired the task. 3/3 tasks are reliably solvable, so admission was not a fluke; 2/3 and 1/3 tasks sit near the reviewer's ability boundary, where repair depends on sampling luck; 0/3 tasks are lucky admissions. per-participant pooled rates range from 75.0% to 100.0%, and all nine tasks outside the 3/3 bucket embed logic or off-by-one defects.

Insights. Since the pooled per-run repair rate estimates the degree to which admission implies reliable solvability, the prevalence of tasks that are successfully admitted with 3/3 Session-C re-runs shows that admission verdict is a stable property of the admitted tasks. Meanwhile, the $\{1/3, 0/3\}$ tail represents the noise floor of the admission filter and flags that tournament turns against those tasks are high-variance events even for a capable Blue. Overall, 79.5% of the sampled tasks are reliably repaired 3/3, the 18.2% boundary mass at 2/3 bounds the share of contested turns whose solvability is partly sampling luck, and the lucky-admission tail is a single task. Admission noise therefore does not materially attenuate the reported Blue success rates.

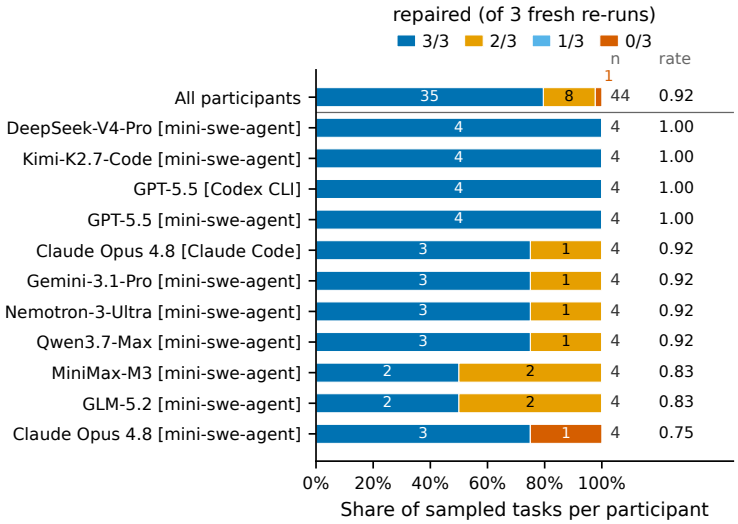


Figure 6: Session-C admission-noise stability study. Each row stacks, as a share of its sampled tasks, how many of the three fresh blind re-runs repaired them (segment labels give absolute counts; n at each bar tip gives the row's sample size); the right column reports the pooled per-run repair rate, and the top row aggregates all 44 tasks. The distribution concentrates on reliable solvability (3/3), with a single lucky admission (0/3).

J Fixed-reference Task Solver

In the main tournament, a participant's Red success mixes the skill of its task author with (i) how difficult its own admitted tasks are and (ii) which solvers its tasks happened to meet. Thus, the match record alone cannot separate these factors. We therefore designate one mid-tier participant as a fixed reference solver: GLM-5.2 with mini-swe-agent, whose contested Blue success of .874 over its 111 in-frame tournament solving runs matches the field aggregate of .851. We have the fixed-reference solver solve everyone of 134 distinct admitted task in the main tournament blind through the exact task-solving evaluation path of a match sub-turn: same Blue prompt, step and wall-clock budgets, and regression and feature test suites, and sealed bugfix scoring. Thus, the task difficulty reading is the same measuring stick for everyone, and is exogenous to the task creation process. Comparing it with the endogenous rate from the main round-robin tournament shows how much of a participant's scheduled Red success transfer to a neutral solver.

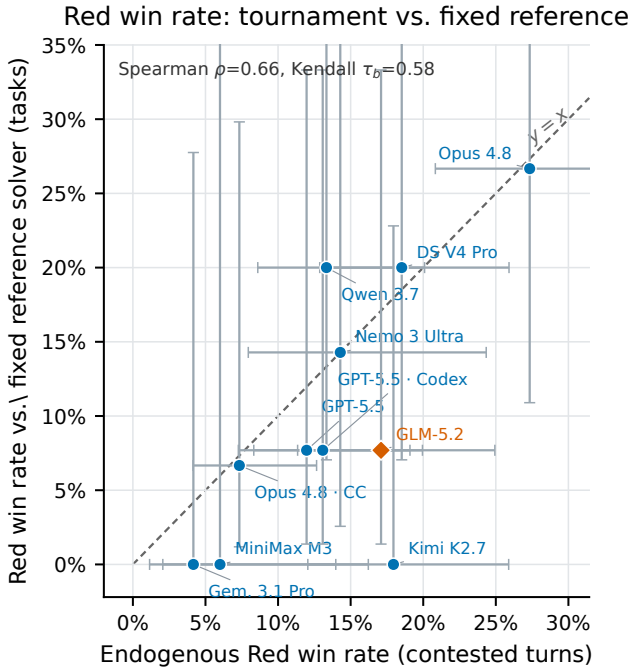


Figure 7: Task difficulty: scheduled field vs. fixed reference. Each point is one participant's Red win rate against the full scheduled solver field (x-axis) and against the fixed reference solver (y-axis). Error bars are Wilson 95% intervals. Points without a harness tag use mini-swe-agent.

Since the reference solver had already solved 111 of the 134 tasks in-match, those turn results are reused for this fixed-reference study rather than re-run. Thus, the 23 tasks remained uncovered by the round-robin main tournament, out of which 13 were authored by the reference itself (admission self-review records are deliberately *not* reused for the reference-authored tasks since those records are the selection event rather than a measurement), and 10 stem from the one pairing involving the reference and the late entrant to the tournament (Gemini 3.1 Pro with mini-swe-agent) that was not part of the active-sampled match-ups in the 50-match main tournament schedule. These 23 get fresh blind runs.

Result. The fixed solver fails to solve 15 of the 134 tasks (11.2%, Wilson 95% $[6.9, 17.6]$), and across the eleven participants the two difficulty scales agree at Spearman $\rho = 0.66$ (Kendall $\tau_b = 0.58$). The strongest task creator of the main tournament keeps the top slot on the fixed-reference scale (Claude Opus 4.8 with mini-swe-agent: 27.3% endogenous, 26.7% reference). Nemotron-3-Ultra stays exactly at 14.3% on both scales. The weakest task generators stay at the bottom (Gemini 3.1 Pro and MiniMax-M3 cause zero fixed-reference solver failures in 10 and 5 tasks, respectively). Kimi-K2.7-Code wins 17.9% of its Red turns against the other participants, yet wins 0% against the fixed-reference solver, so its success exploits weaknesses of the other participants rather than its intrinsic difficulty; conversely, Qwen3.7-Max (13.3% $\rightarrow$ 20.0%) and DeepSeek-V4-Pro (18.5% $\rightarrow$ 20.0%) author tasks that are *harder* for the fixed-reference solver than the other participants. Table 3 reports the per-participant comparison and Figure 7 the scatter plot visualization of it.

Insights. The moderate-to-strong positive rank agreement between the main round-robin tournament and fixed-reference evaluation (Spearman $\rho = 0.66$, and Kendall $\tau_b = 0.58$) means the endogenous difficulty ordering is substantially a property of the *generated tasks* rather than the resultant of which solvers a task author happened to meet. One caveat: with 5–15 tasks per participant, the confidence intervals are wide. Our future extension plans more in-depth study of the finer-grained ordering that supports rank-level agreement claims.

K SCALING TASK DIFFICULTY WITH TASK GENERATOR CAPABILITY

While the fixed reference solver of Appendix J removes the solver-field confound, it still compares the participants of different origin that happen to be in the field. Thus, their task difficulties are a property of the generated tasks, not that it changing with task authoring capability. We therefore hold the solver fixed and move the *task author* along a known capability axis. Successive versions of one model family are exactly such an axis, so we built a three-rung ladder from GLM-5.1 $\rightarrow$ GLM-5.2 $\rightarrow$ GLM-5.3, all running the same mini-swe-agent harness. Each version authored tasks over the same repository field under the identical task generation protocol and validation gates. GLM-5.1 and GLM-5.3 wrote fresh banks (12 and 14 admitted tasks, respectively); the 13 GLM-5.2 tasks are its admitted tasks from the main tournament. We use DeepSeek-V4-Pro with mini-swe-agent as the fixed reference solver for this study, as it is another mid-tier solver whose contested Blue success of .919 sits near the field aggregate of .851, and a model outside the ladder family. The fixed-reference solver than then blind-solved every one of the 39 ladder tasks through the exact task-solving path of a match sub-turn: same Blue prompt, step and wall-clock budgets, regression and feature test suites, and sealed bugfix oracle. Since the fixed-reference solver had already solved the 13 GLM-5.2 tasks in-match during the main round-robin tournament, those turn records are reused as part of this study.

Result. Solver failure rises with the capability frontier (Table 2): $1/12$ (8.3%, Wilson 95% $[1.5, 35.4]$) for GLM-5.1, $0/13$ for GLM-5.2, and $6/14$ (42.9%, $[21.4, 67.4]$) for GLM-5.3. The upward trend across the ladder is significant (exact conditional Cochran–Armitage trend test over the 39 draws, z=2.32, one-sided p=.015), and the newest version is clearly separated from the pooled older ones ($6/14$ vs. $1/25$, Fisher p=.005), while the two older versions are indistinguishable from each other. All seven failures are complete collapses (Blue composite 0.00), and the solver works *harder* on the tasks it loses (mean 37.3 vs. 31.5 steps), so failures are exhausted budgets ending in broken behavior rather than early give-ups.

Composition vs. difficulty. The defect-category mix shifts along the ladder (security vulnerabilities $4 \rightarrow 2 \rightarrow 7$; logic errors $2 \rightarrow 9 \rightarrow 2$), and GLM-5.2's bank is dominated by logic errors that the solver always repaired. Composition alone, however, does not explain the jump: within security-vulnerability tasks only, the older versions' tasks never defeat the solver ($0/6$) while GLM-5.3's do in $5/7$ (Fisher p=.016) even though that category was among the most solvable for the full field in main tournament (84.9% Blue success). Admission yield does not explain it either: per-attempt yields are $12/39$ (30.8%), $13/32$ (40.6%), and $14/37$ (37.8%), non-monotonic across the ladder, with the difficulty jump occurring where yield slightly *falls*. Thus, difficulty rises alongside rather than because of other task generation statistics.

Head-to-head tournament among the ladder versions. The fixed solver sits outside the family, so it cannot say how the three versions fare against one another. Thus, we conducted a complete round-robin tournament where every version plays Red and Blue on every repository once per pairing (Figure 4). The head-to-head tournament produces endogenous match-ups which correlated with the ladder. GLM-5.3's tasks defeat GLM-5.1 in $7/14$ contested turns and GLM-5.2 in $4/14$, while the two older versions almost never defeat anyone ($3/50$ pooled across their four cells; one-sided exact Fisher against GLM-5.3's $11/28$: p=.0004). The newest version is also the strongest solver, conceding $2/25$ contested turns to its siblings' tasks versus $4/26$ for GLM-5.2 and $8/27$ for GLM-5.1; it won both of its matches (7–1 and 4–1 in Red wins) and GLM-5.2 won the third (1–0), a clean 2–1–0 sweep of the standings. The sibling readings agree with the fixed-reference solver where they overlap: the outside solver failed on $6/14$ of GLM-5.3's tasks, the siblings on $4/14$ and $7/14$, and on the two older versions' tasks no solver in either study concedes more than a single turn. Siblings share a training lineage and may share blind spots, so the matrix corroborates rather than replaces the fixed-solver measurement.

Insights. With solver, harness, repositories, and protocol held fixed, tasks authored by the newest version of the family are an order of magnitude harder for the same solver than the pooled older versions' tasks (42.9% vs. 4.0% failure). Together with the trend test, this is direct evidence for the self-scaling property of SWE-DUEL design: as the authoring agents improve, the tasks they produce get harder for a fixed solver. One fixed-reference solver (DeepSeek V4 Pro with mini-swe-agent) is the single measuring stick just like Appendix J where the rank agreement of ρ=.66 suggests the reading transfers to other mid-tier solvers, and the within-family round-robin (Figure 4) supplies two sibling solvers whose failure rates climb the same way, but siblings share the family's training

lineage, so they corroborate rather than independently re-measure the ladder. The ladder has three rungs from one model family; such ladders are cheap to extend (the fresh rungs here cost $15.27 total), so the task difficulty scaling trend can be continuously verified as new versions ship. At 12–14 tasks per level and one draw per task at temperature 0.2, adjacent-version resolution is limited (the admission stability study in Appendix I measured a .917 per-run repair rate, and the two older levels' intervals overlap almost entirely), so even though the task difficulty scaling is not strictly per-version monotone, the data still supports it roughly scaling with the capability frontier.

L Role Decomposition and Contested-only BT Refits

Role Decomposition. Table 8 separates the two capabilities that the end-to-end match score intentionally combines: Red task generation capability, and Blue task solving capability. Thus, we analyze the task solving-only BT fit of the main round-robin tournament. Coverage is the number of admitted tasks divided by the 15 scheduled repository slots; an unsuccessful chain of 5 attempts counts as one exhausted slot. The task-solving intervals are task-clustered bootstrap intervals. Figure 8 decomposes each system into task creation and task solving. Nemotron-3-Ultra's contested Red win rate is $10/70$, but nine of the ten wins come from one of its seven admitted tasks. A bootstrap draw that omits both winning clusters rates the task creator at zero while a draw that includes the nine-win task several times is rated near 0.5. Thus, the result is the field's widest task-creation interval, $[0.000, 0.450]$. Appendix O analyzes the responsible task in detail.

Table 8: Task solving and generation-coverage decomposition for the headline tournament. Participants are shown in the order of full-schedule tournament's BT ratings. Solving BT is a role-conditioned fit on contested Blue turns only. Coverage is admitted slots divided by the 15 repository slots.

Full rank	Competitor	Solving BT	95% CI	Solving success	Admitted / scheduled slots	Coverage
1	GPT-5.5 [codex]	+2.19	[+1.35, +4.89]	98.3%	13 / 15	86.7%
2	Claude Opus 4.8 [mini-swe-agent]	-0.32	[−0.81, +0.08]	85.7%	15 / 15	100.0%
3	GPT-5.5 [mini-swe-agent]	+1.45	[+0.82, +2.82]	96.4%	13 / 15	86.7%
4	Kimi-K2.7-Code [mini-swe-agent]	+0.09	[−0.42, +0.59]	88.3%	13 / 15	86.7%
5	DeepSeek-V4-Pro [mini-swe-agent]	+0.25	[−0.28, +0.79]	89.9%	15 / 15	100.0%
6	GLM-5.2 [mini-swe-agent]	+0.01	[−0.42, +0.47]	87.4%	13 / 15	86.7%
7	Qwen3.7-Max [mini-swe-agent]	+0.22	[−0.29, +0.76]	89.0%	15 / 15	100.0%
8	Claude Opus 4.8 [claude-code]	-0.23	[−0.71, +0.16]	83.9%	15 / 15	100.0%
9	Nemotron-3-Ultra [mini-swe-agent]	-0.87	[−1.46, −0.49]	75.6%	7 / 15	46.7%
10	MiniMax-M3 [mini-swe-agent]	-1.12	[−1.64, −0.84]	70.5%	5 / 15	33.3%
11	Gemini-3.1-Pro [mini-swe-agent]	-1.69	[−2.65, −1.08]	61.8%	10 / 15	66.7%

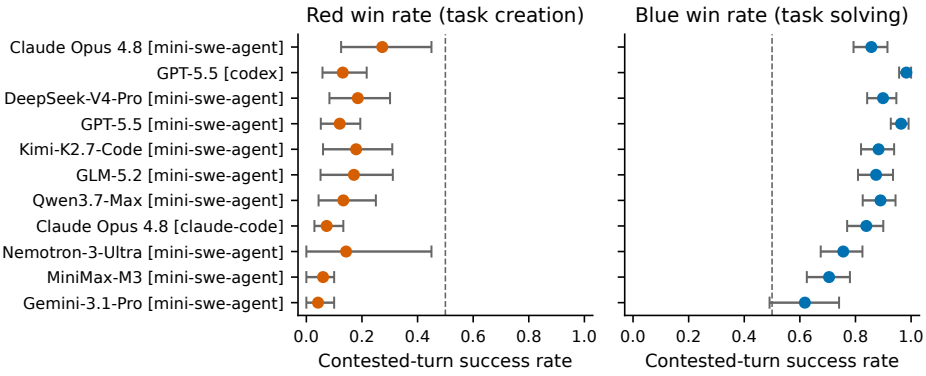


Figure 8: Task-clustered contested task-creation and task-solving success by competitor. Missing-Red auto-wins are excluded.

On preventing saturation. As we have seen in this section, models are generally better at finding bugs and solving tasks than they are at authoring tasks. Plus, the admission of a task is contingent upon the task author's task solving capability. Thus, the shrinking gaps in task solving capabilities

among frontier models could shrink the gaps in their task authoring capabilities, potentially leading to saturation of the arena. Two solutions (one short-term, and one long-term) address this problem:

- **(Short-term, temporary)** Appendix C shows the 15 repositories used in our experiments. They are selected for being relatively small-sized projects in their respective languages/domains. This is because we need their test suites to be lightweight enough for quick execution of validation gates for the sake of our experiments, and for the agent participants to not face too much difficulty with generating tasks and burn the expensive LLM API resources (Appendix R). Thus, practitioners adopting SWE-DUEL may extend to cover bigger repositories with more complex codebases that has wider surface area for feature generation and more subtle and nuanced bug embedding. This would help better highlight the gap in coding capabilities of different frontier models, and prevent saturation of the arena that would otherwise be confined to the same set of 15 lightweight repositories.
- **(Long-term, permanent)** A future extension of this work could focus on the solvability checks that does not require blind self-reviewing by the task author, and thus is not capped by the task solving capability of the task author. For example, the solvability check gate could involve the privileged information such as informing the self-review session about the existence of the bug, or to look for the specific type of the bug. This would still prove the bug is fair and solvable, but now the task author has more leeway to embed more subtle and challenging bugs that no longer needs to be detected and fixed by the blind self-review session.

Contested-only refit. The contested-only refit moves GPT-5.5 with mini-swe-agent from third to first, GPT-5.5 with Codex from first to second, Claude Opus 4.8 with mini-swe-agent from second to fifth, Nemotron-3-Ultra from ninth to third, and MiniMax-M3 from tenth to seventh. The corresponding durable-slot coverage rates are $5/15$ (33.3%) for MiniMax-M3, $7/15$ (46.7%) for Nemotron-3-Ultra, and $10/15$ (66.7%) for Gemini-3.1-Pro; the table reports all participants. The rank moves are dominated by the task generation coverage penalty that the full-schedule estimand imposes via automatic wins. In Nemotron-3-Ultra's case, with $7/15$ slots admitted, each of its ten matches concedes eight uncontested Blue points, totalling 80 auto-win turns across the main tournament. The contested-only refit deletes exactly those 80 concessions, and re-weights records to the higher win-rate contested-only Red wins. Thus, its refit coefficient ($+0.145$) is the third highest, jumping in ranking from ninth to third. The position is nonetheless fragile as its clustered interval $[-0.35, +0.75]$ zero at its lower bound, and nine of its ten contested Red wins arise from a single admitted task. Appendix O dissects that task, and its ten solving runs.

M RED–BLUE TRANSFER

Figure 9 depicts the Red/Blue success rates between any two participants from the main round-robin tournament. Each non-diagonal cell has one scheduled turn per repository in the main tournament, so the displayed $x/15$ values should be read as raw event counts, not as precise estimates. Some cells have no turns due to the late entrant (Gemini 3.1 Pro with mini-swe-agent) only having active-sampled matchups with a subset of the existing participant pool. A future Red–Blue transfer study should add a new axis by freezing a common task subset per task author before comparing solvers.

N TURN-LEVEL BRADLEY–TERRY ANALYSIS

The match-level estimand in Section 3.4 compresses each match into a single outcome. This section reports the complementary turn-level estimand, which keeps every turn in a match as independent matches: each turn contributes one Red-versus-Blue Bradley–Terry outcome under the same draw-as-half-win convention, so the BT fit measures per-turn strength rather than per-match win aggregation. We fit two variants on the main round-robin tournament's $1{,}500$ persisted turn records ($1{,}219$ contested turns and 281 missing-Red auto-wins. The *full-schedule* variant keeps all turns and scores an auto-win as a Blue win. The *contested-only* variant drops the 281 auto-win turns.

Table 9 reports the point estimates and 95% intervals for both BT ratings, and tournament rankings. We perform $1{,}000$ bootstrap draws to resample the generated tasks. rather than individual turns because one task is defended by many participants, and those turns are correlated through the shared task. The auto-win turns due to empty Red–repository slots form a single cluster, since they are

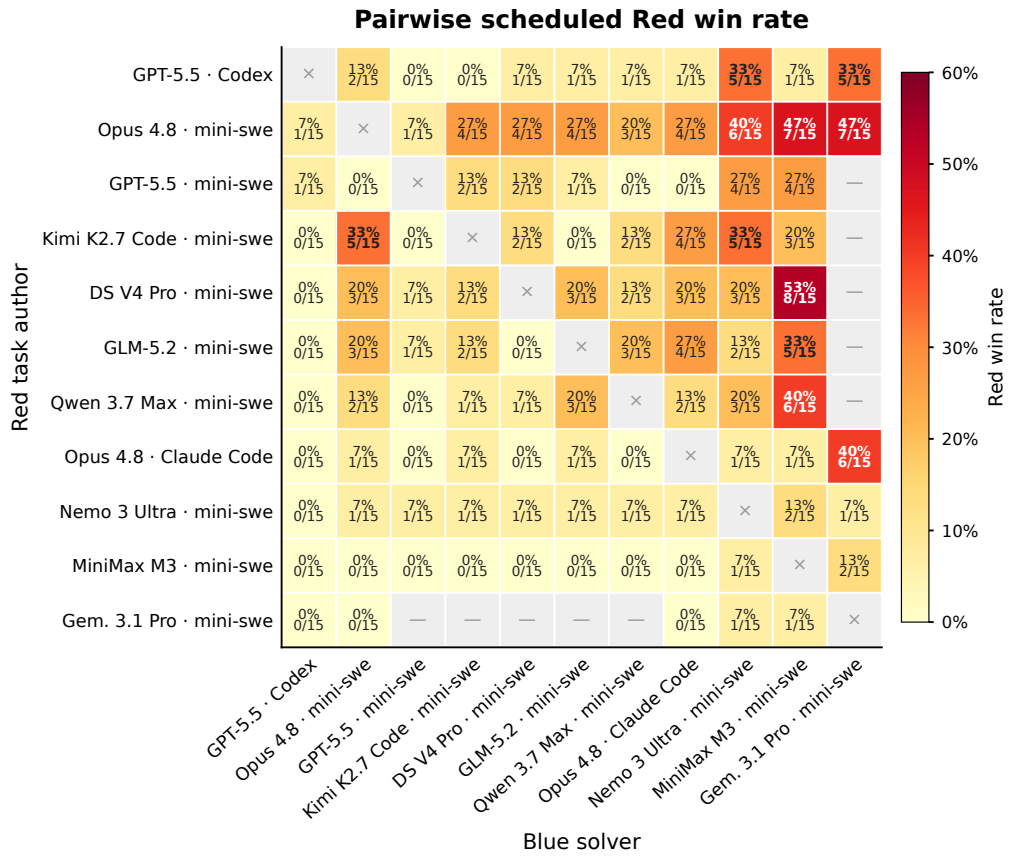


Figure 9: Red–Blue transfer matrix, ordered by the tournament's Bradley–Terry log-strength rating. Each observed cell reports the Red win rate as a percentage and as raw wins over the 15 scheduled repository turns ($n = 15$).

repeated concessions of one generation failure rather than independent events. In the full-schedule variant, every participant's turns is balanced by design (fifteen Red and fifteen Blue turns per match), whereas contested-only turns scales with admitted bank size (103–269 turns). Appendix L further analyzes the different effects by automatic wins, and contested-only turns in the tournament.

Auto-win policy, not granularity, drives estimand sensitivity. The full-schedule turn-level rankings is much closer to the main tournament's full-schedule match-level ranking (Kendall's $\tau = 0.782$) than to the contested-only ordering ($\tau = 0.382$), with the contested-only ordering itself at $\tau = 0.527$ from the match ordering.

Where the rankings change. In the full-schedule turn-level BT rankings, the ranking changes are confined to the top seven, whose intervals all overlap since the full-schedule main tournament: the match-level leader GPT-5.5 with Codex drops to fourth ($+0.18$), conceding 2×10 auto-win turns from its $13/15$ coverage, while the two full-coverage strong task creators rise: Claude Opus 4.8 with mini-swe-agent to first ($+0.30$) and DeepSeek-V4-Pro to second ($+0.25$). This reflects the per-turn weight on Opus's field-best 27.3% Red win rate. The bottom four positions are identical to the main tournament's full-schedule match ordering (Opus with Claude Code, Nemotron-3-Ultra, MiniMax-M3, Gemini-3.1-Pro). MiniMax-M3 ($[-0.84, -0.01]$ conceded 100 auto-win turns from its five admitted slots), and Gemini ($[-0.98, -0.12]$, conceded five slots combined with the field's lowest contested Blue success (Figure 8). Nemotron-3-Ultra's $[-0.70, +0.19]$ still spans zero despite 80 concessions, because the single task that concentrates its Red wins (Appendix O) keeps its contested record strong.

Table 9: Turn-level Bradley–Terry ratings for the main tournament. BT values are point estimates with 95% percentile intervals from 1,000 task-cluster bootstrap draws; rank intervals are central 95% intervals over the same draws. The full-schedule variant treats missing-Red auto-wins as Blue wins; the contested-only variant drops them.

Competitor	Full-schedule turn BT (95% CI)	Full-schedule 95% rank interval	Contested-only turn BT (95% CI)	Contested-only 95% rank interval
Claude Opus 4.8 [mini-swe-agent]	$+0.30$ $[-0.10, +0.80]$	1–8	$+0.12$ $[-0.34, +0.61]$	1–10
DeepSeek-V4-Pro [mini-swe-agent]	$+0.25$ $[-0.15, +0.73]$	1–8	$+0.07$ $[-0.31, +0.54]$	1–10
GPT-5.5 [mini-swe-agent]	$+0.19$ $[-0.20, +0.72]$	1–9	$+0.17$ $[-0.28, +0.66]$	1–9
GPT-5.5 [codex]	$+0.18$ $[-0.20, +0.73]$	1–9	$+0.15$ $[-0.29, +0.84]$	1–9
Kimi-K2.7-Code [mini-swe-agent]	$+0.17$ $[-0.23, +0.68]$	1–9	$+0.13$ $[-0.28, +0.62]$	1–10
Qwen3.7-Max [mini-swe-agent]	$+0.14$ $[-0.28, +0.65]$	1–9	-0.05 $[-0.46, +0.46]$	1–10
GLM-5.2 [mini-swe-agent]	$+0.14$ $[-0.29, +0.67]$	1–9	$+0.10$ $[-0.33, +0.59]$	1–10
Claude Opus 4.8 [claude-code]	-0.10 $[-0.50, +0.34]$	2–10	-0.33 $[-0.74, +0.14]$	4–11
Nemotron-3-Ultra [mini-swe-agent]	-0.26 $[-0.70, +0.19]$	4–11	$+0.14$ $[-0.35, +0.75]$	1–10
MiniMax-M3 [mini-swe-agent]	-0.44 $[-0.84, -0.01]$	6–11	$+0.10$ $[-0.39, +0.60]$	1–10
Gemini-3.1-Pro [mini-swe-agent]	-0.57 $[-0.98, -0.12]$	7–11	-0.60 $[-1.06, -0.10]$	7–11

Rank uncertainty is larger in turn-level BT. Turn-level analysis provides a more conservative treatment of ranking confidence intervals than match-level. This is to be expected as we have observed that participants are generally better at solving tasks than generating difficult tasks, thus every participant has a non-trivial number of Blue wins that moderates their BT ratings.

O A Concentrated Red-win Record

Appendix L singles out Nemotron-3-Ultra for having the largest contested-only rank gain and the widest clustered Red win rate interval. Nine of its ten contested Red wins arise from one admitted artifact, task `71a623dd` on the Go `gorilla/csrf` repository. This section analyzes the artifact of the task and its ten persisted solving runs to characterize what its difficulty measures.

Task details. The feature adds a `SkipPaths` option: request paths equal to a configured path skip CSRF validation exactly, and paths beneath a configured path ending in a slash skip by prefix. The sealed defect is a one-condition change in the prefix matcher: the request path must additionally be strictly longer than the cleaned skip path, so a request equal to the configured prefix itself (for instance `POST /webhook/` under skip path `/webhook/`) fails validation instead of being skipped. The sealed bug test requires the lenient reading; no visible signal does. The visible feature suite enforces the strict behavior for the unslashed path (`/webhook` must be rejected) and accepts only strict descendants, and the PR's own doc comment over-claims the lenient semantics, contradicting those tests. A solver must therefore settle a specification conflict in which every visible signal corroborates the defective behavior.

The ten solving runs. One participant solved the task (GPT-5.5 with Codex CLI) while the other nine lost. Each of the 9 losses is a latent bug ship (existing and feature suites pass, only the sealed suite fails), and one of the nine also appears in the audit sample of Appendix G as a genuine miss (solving run `56210ac0`). Eight of the nine explicitly inspected or reasoned about the strict-length condition; five filed low- or info-severity findings that framed the discrepancy as a documentation error rather than a functional one. Only one solver (GLM-5.2) never mentions the matching condition at all. The single success is the one solving run that separated the two path forms (the unslashed path must stay protected while the prefix path itself must skip) and added its own matcher tests; it too labeled the issue low-severity, but repaired it.

The task contains a genuine bug. The manual audit reveals the sampled solving run is a true failure of the participants in identifying the bug, rather than an unsound bug-test oracle. But it does not test whether *Blue* solver can locate the hidden bug, as most solvers located the exact condition according to the manual inspection of the Blue agents' reasoning traces. Consequently, Nemotron-3-Ultra's third-place contested refit is heavily influenced by this single task for its Red win rate estimand. This concentration reveals an interesting observation information about the task author: five of its seven admitted tasks defeat no one, so its bank is one exceptionally well-constructed defect plus six ordinary tasks. Thus, the clustered intervals are high variance for Red capability estimand for Nemotron-3-Ultra participant.

P Defect Composition

Bug types are categorized using the label given the *Red* agents, and persisted in task metadata. They include logic, security, off-by-one, type-confusion, resource, input-validation, and race-related defects.

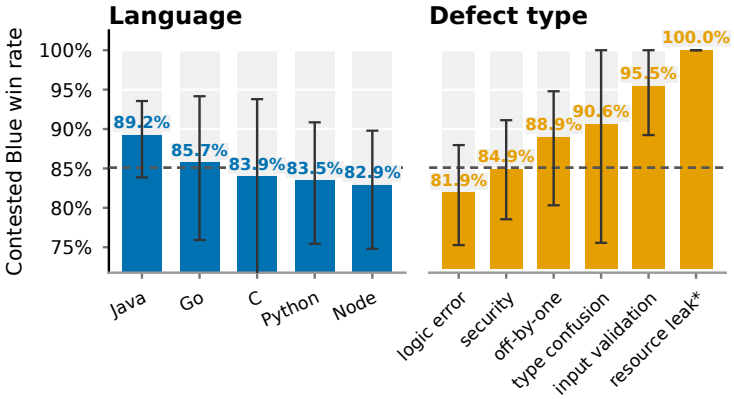


Figure 10: **Contested Blue win rates by language (left) and defect type (right)** with 95% task-clustered bootstrap intervals. The resource-leak bar (*) is based on one task (nine turns) and is not interpreted. Missing-Red auto-wins are excluded.

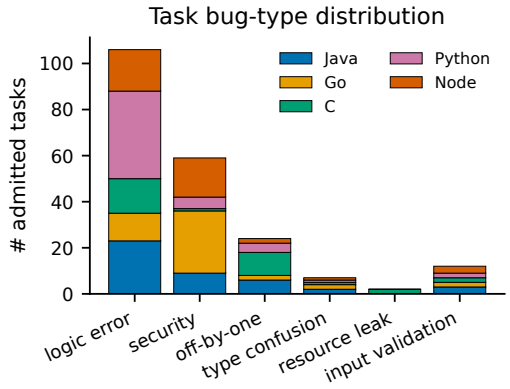


Figure 11: Distribution of admitted tasks across the observed defect taxonomy. Taxonomy frequency describes the generated bank and should not be interpreted as the prevalence of defects in human software.

Q Harness Ablation Study

In the harness tournament, we select three harnesses (mini-swe-agent, Codex CLI, and Claude Code), and three models (DeepSeek-V4-Pro, GLM-5.2, or Kimi-K2.7-Code). Each of the selected model creats tasks under all three selected harnesses on all 15 repositories. Thus, each harness participant pools the three base models (15-task slots each) into a single strength per harness of 45 tasks. We conducted a round-robin tournament where each model plays against same-model but different-harness participant. Thus, no cross-model sub-match contributes to this tournament. The BT column of Table 10 presents the match-level Bradley–Terry model refit with the models-aggregated harnesses as players. Each row of Table 10 represents the performance of each harness aggregated over three base models. Since there are 3 harnesses selected for this tournament, each harness gets 2 matchups, giving $45 \times 2 = 90$ Red turns for each harness throughout the harness–ablation tournament. Involving Blue turns as well the opposite direction, each harness records 90 Red + 90 Blue = 180 total turns throughout the harness ablation tournament.

Statistical resolution. Since turns are decisive Red–Blue outcomes, we derive the 95% CI for score rate with using clustered Bernoulli draws. The three aggregate point estimates span 0.03 score-rate points while each interval spans about 0.17, and even the largest within-model contrast (DeepSeek V4 Pro: Codex CLI $0.62\,[0.46, 0.78]$ vs. Claude Code $0.40\,[0.27, 0.57]$) overlaps on $[0.46, 0.57]$. Table 11 shows the head-to-head match records between each harness pair, and its the score margins corroborates the point estimates in Table 10: $+2\,[-16, +22]$, $+2\,[-16, +20]$, and $-8\,[-28, +12]$ points out of 90 paired turns, so paired differences below roughly 0.2 score-rate points are undetectable at this budget. Figure 12 breaks down the performance of each harness when controlled by base model. Within every model the three intervals overlap, so just like models-aggregated statistics, model-controlled bar crossings are descriptive, and cannot establish a per-model harness ranking.

Table 10: Harness ablation aggregated over three fixed base models. Score sums both roles over the full schedule; Rate is the score rate with a 95% task-clustered percentile interval; Red win and Blue win are per-role turn success rates over the same full schedule. BT is the match-level Bradley–Terry log-strength fit to the nine same-model sub-match outcomes pooled across the base models, with a 95% percentile interval from the ordinary match bootstrap used by the headline rating.

Harness	W	D	L	Score	Rate (95% CI)	Red win	Blue win	BT (95% CI)
mini-swe-agent	4	1	1	93/180	0.52 [0.43, 0.60]	0.13	0.90	+0.73 [−0.25, +5.56]
Codex CLI	2	1	3	90/180	0.50 [0.42, 0.60]	0.11	0.89	−0.25 [−3.50, +1.34]
Claude Code	1	2	3	87/180	0.48 [0.40, 0.58]	0.09	0.88	−0.49 [−3.83, +0.65]

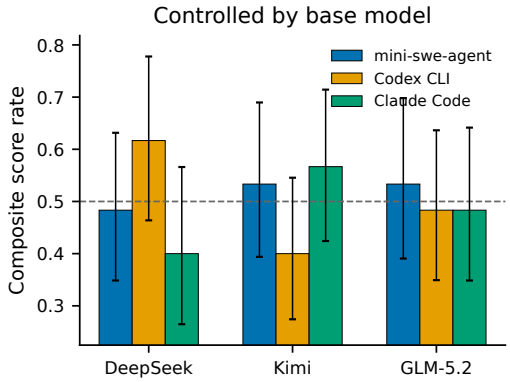


Figure 12: Harness ablation controlled by base model. Composite two-role score rate for each fixed base model under each of the three harnesses; whiskers are 95% task-clustered percentile intervals (60 turns, 45 clusters per bar).

Harness effect vs. model–harness fit. The two CLI harnesses are prompt- and tool-engineered around their providers' model families, whereas mini-swe-agent is a minimalist harness without a prompt tuned for specific models. In the ablation, all three are cross-paired with DeepSeek V4 Pro, GLM 5.2, and Kimi K2.7 Code. Thus, there are no native (unfair) pairings that could confound harness effect analysis. Native pairings occur only in the main tournament (GPT-5.5 with Codex CLI and Claude Opus 4.8 with Claude Code) where we just want to observe their performances for the sake of curiosity, and not meant for harness probe study.

R Cost

Cost is partitioned into Red generation, blind self-review validation, and Blue evaluation. A successfully created task includes the costs previous failed attempts within the same slot. The full-schedule also counts costs of all the failed attempts in exhausted slots that never appear in a contested turn. Table 12 shows the breakdown of costs by generation, self-review validation, and blue evaluation for each participant throughout the main round-robin tournament, and Figure 13 shows the scatter plot

Table 11: Aggregate score for each harness pair. Margin is (A−B) points with a 95% clustered percentile interval on the paired per-turn difference. The direct comparisons resolve no ordering at this sample size and the apparent cycle in pair totals is consistent with sampling noise.

Harness A	Harness B	A score	B score	Margin (95% CI)	Models
Claude Code	Codex CLI	46	44	$+2\,[-16,\,+22]$	3
Codex CLI	mini-swe-agent	46	44	$+2\,[-16,\,+20]$	3
Claude Code	mini-swe-agent	41	49	$-8\,[-28,\,+12]$	3

Table 12: Generation, blind self-review validation, and Blue evaluation API cost by competitor. Generation cost includes failed attempts in slots that never produced a task, preventing successful-only accounting.

Competitor	Gen $	Val $	Eval $	Total $
GPT-5.5 [codex]	149.56	47.54	444.79	641.89
GPT-5.5 [mini-swe-agent]	48.38	17.15	181.65	247.18
Claude Opus 4.8 [mini-swe-agent]	44.93	22.12	156.54	223.59
Gemini-3.1-Pro [mini-swe-agent]	106.18	19.77	61.81	187.76
Claude Opus 4.8 [claude-code]	42.31	14.04	122.70	179.05
Nemotron-3-Ultra [mini-swe-agent]	84.66	2.85	83.82	171.34
Qwen3.7-Max [mini-swe-agent]	36.44	12.40	58.38	107.22
GLM-5.2 [mini-swe-agent]	43.51	7.14	54.32	104.97
Kimi-K2.7-Code [mini-swe-agent]	34.85	7.16	50.38	92.39
MiniMax-M3 [mini-swe-agent]	46.51	2.13	38.12	86.77
DeepSeek-V4-Pro [mini-swe-agent]	32.36	6.04	28.67	67.07

visualizing the performance of the participants by total output tokens and API cost (in USD). GPT-5.5 with Codex cost $641.89, 2.6 times the next-highest total and 7.4 times MiniMax-M3. These totals also vary with generation yield (some participants taken multiple attempts to generate a task, causing the cost to increase), and number of matches (the Gemini 3.1 Pro late entrant plays only five active-sampled matches).

Turn-level Blue spend. Table 13 breaks down the main tournament's $1{,}219$ contested turns into each participant's Blue turns. Participant totals range from 0.64M to 1.58M output tokens, and turn medians from 5.9k to 11.1k. Nine of eleven correlation intervals include zero. Nemotron-3-Ultra has a positive association ($r = .384$ [.213, .532]), whereas GPT-5.5 with mini-swe-agent has a small negative one ($r = -.107$ [-.197, -.004]). Difficult tasks can induce longer trajectories in the *Blue* agent, while easier tasks can be solved in shorter trajectories. Thus, this analysis is descriptive and exploratory.

Output-token-matched task-solving ranking. As a supplementary sensitivity analysis, we cap every participant at 638,976 recorded output tokens, the smallest available participant total. In each of 1,000 draws we randomly order that participant's complete-case contested solving runs, admit turns without replacement while the cap permits, and refit the role-conditioned model $P(\text{Blue wins}) = \text{sigmoid}(d_{\text{Blue}} - a_{\text{Red}})$. Table 14 reports medians and 95% intervals over this sub-sampling distribution. GPT-5.5/Codex and GPT-5.5/mini-swe remain first and second; the matched order is close to the complete-case solving order ($\tau = 1.000$).

S Newcomer Onboarding and Marginal Cost

The main tournament admitted one late entrant, Gemini-3.1-Pro with mini-swe-agent, after the initial ten-player round-robin had completed. The entrant followed the identical task creation, validation gates, and self-review procedure to author its own bank over the same 15 repository slots, then sampled $\lceil \log_2 N \rceil + 1 = 5$ matches out of $N = 10$ incumbents, using the active-sampling algorithm presented by Chatbot Arena (Chiang et al., 2024) (Section 3.4). Ten of its 15 slots were admitted; the

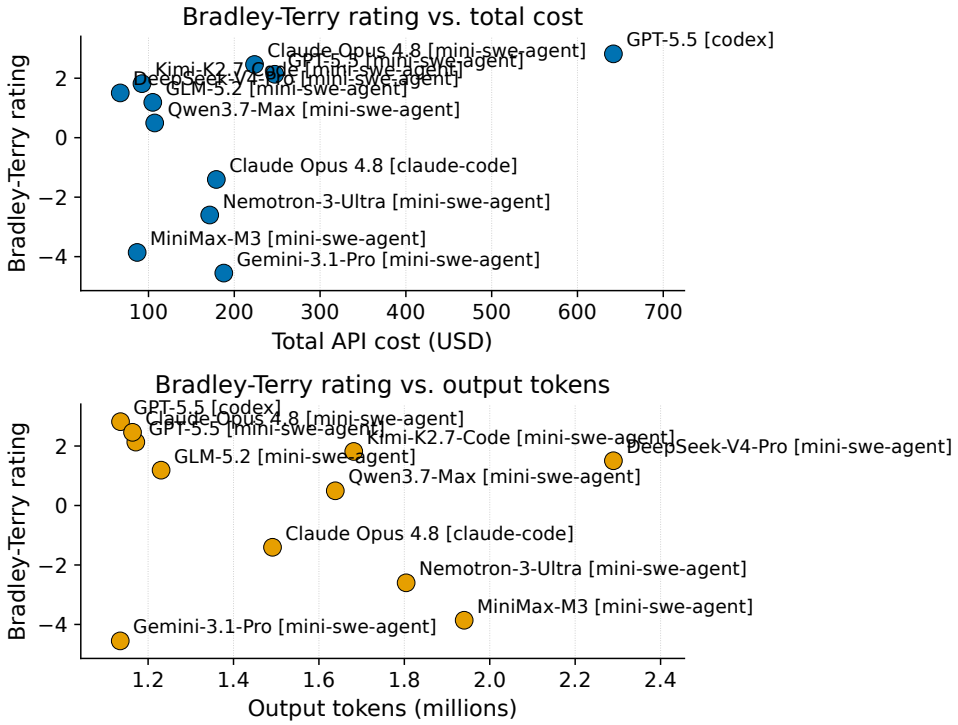


Figure 13: Secondary efficiency diagnostic relating the tournament's Bradley–Terry log-strength rating to API cost and output-token use. These observational associations combine model pricing, generation coverage, and behavior; they are not a controlled efficiency comparison.

Table 13: Contested Blue task-solving resource use. "Turns" is the number of all contested Blue turns for a given participant. "Success" is the rate of successful task solving out of all turns; "Output M" is the total output tokens over all contested turns, in millions; "Median k" is the median per-turn output tokens, in thousands. The final column is Pearson's correlation between binary task-solving success and $\log(1 + \text{output tokens})$, with a task-cluster bootstrap 95% interval.

Competitor	Turns	Success	Output M	Median k	r [95% CI]
Claude Opus 4.8 [mini-swe-agent]	119	0.857	0.75	5.9	-0.04 $[-0.21, +0.15]$
Claude Opus 4.8 [claude-code]	118	0.839	0.80	6.9	-0.18 $[-0.35, +0.00]$
GPT-5.5 [mini-swe-agent]	111	0.964	0.84	7.3	-0.11 $[-0.20, -0.00]$
GPT-5.5 [codex]	120	0.983	0.81	6.1	$+0.09$ $[+0.00, +0.18]$
DeepSeek-V4-Pro [mini-swe-agent]	109	0.899	1.15	10.1	-0.01 $[-0.23, +0.21]$
GLM-5.2 [mini-swe-agent]	111	0.874	0.78	6.6	$+0.04$ $[-0.14, +0.22]$
MiniMax-M3 [mini-swe-agent]	129	0.705	1.58	11.1	-0.07 $[-0.20, +0.08]$
Kimi-K2.7-Code [mini-swe-agent]	111	0.883	1.05	8.6	-0.04 $[-0.25, +0.18]$
Qwen3.7-Max [mini-swe-agent]	109	0.890	0.84	7.0	-0.12 $[-0.30, +0.09]$
Nemotron-3-Ultra [mini-swe-agent]	127	0.756	1.37	10.2	$+0.38$ $[+0.21, +0.53]$
Gemini-3.1-Pro [mini-swe-agent]	55	0.618	0.64	8.6	$+0.01$ $[-0.25, +0.26]$

five exhausted slots gives missing-Red auto-wins for its opponents exactly as for incumbents. Nothing about the completed tournament is recomputed: incumbent banks, the 45 completed matches, and all cached solving runs are reused unchanged, and the Bradley–Terry fit simply extends.

Marginal entrance cost. Table 15 juxtaposes the main tournament's $2,109.21 total API spend and the late entrant's costs. The marginal entrance cost of $243.40 comprises $125.95 of task creation (generation cost of $106.18 and $19.77 of blind self-review validation, including every failed attempt in the entrant's slots) and $117.45 of match evaluation: $61.81 for the entrant's 55 fresh solving

Table 14: Output-token-budget-matched contested task-solving ranking at 638,976 tokens per participant. This task-solving-only estimand excludes Red task creation, uses provider-reported output tokens, and is not meant to replace the main tournament ranking.

Rank	Competitor	Solving BT	95% interval	Rank interval	Median turns
1	GPT-5.5 [codex]	+2.60	$[+2.40, +4.94]$	1–1	94
2	GPT-5.5 [mini-swe-agent]	+1.38	$[+0.97, +2.41]$	2–2	84
3	DeepSeek-V4-Pro [mini-swe-agent]	+0.17	$[-0.32, +0.86]$	3–7	61
4	Qwen3.7-Max [mini-swe-agent]	+0.15	$[-0.21, +0.57]$	3–6	83
5	Kimi-K2.7-Code [mini-swe-agent]	+0.02	$[-0.44, +0.56]$	3–7	68
6	GLM-5.2 [mini-swe-agent]	-0.08	$[-0.42, +0.28]$	3–7	91
7	Claude Opus 4.8 [mini-swe-agent]	-0.36	$[-0.71, -0.07]$	5–8	101
8	Claude Opus 4.8 [claude-code]	-0.39	$[-0.73, -0.09]$	6–8	86
9	Nemotron-3-Ultra [mini-swe-agent]	-0.98	$[-1.44, -0.48]$	8–10	60
10	MiniMax-M3 [mini-swe-agent]	-1.18	$[-1.63, -0.68]$	9–10	53
11	Gemini-3.1-Pro [mini-swe-agent]	-1.75	$[-2.07, -1.56]$	11–11	55

runs on incumbent tasks and $55.64 for the incumbents' 48 recorded solving runs on the entrant's 10 admitted tasks.

Table 15: API-cost ledger: the main tournament's cost split and the marginal cost of the late entrance. Shares are fractions of the tournament total; per-participant totals are in Table 12.

Cost component	USD	Share
Main tournament		
Task creation (incl. failed attempts)	669.69	31.8%
Blind self-review validation	158.34	7.5%
Blue evaluation	1281.18	60.7%
Tournament total	2109.21	100.0%
Entrance: Gemini-3.1-Pro [mini-swe-agent]		
Bank authoring (task creation + validation)	125.95	6.0%
Entrant solving of incumbent tasks	61.81	2.9%
Incumbents solving entrant tasks	55.64	2.6%
Marginal entrance total	243.40	11.5%

Late entrant placement. The five-match entrance resolved a coherent placement. Gemini-3.1-Pro finished with W/D/L record of 0–1–4, and the full-schedule BT point estimate (-4.55, 95% CI $[-12.57, -2.45]$), ranking last with a bootstrap median rank of 11 (central 95% interval 9–11). Its contested solving success is $.618$ $[.491, .741]$, the lowest of the participant pool, and its contested Red win rate $.042$. Refitting BT on the 45 incumbent-only matches reproduces the incumbent ordering of the full 50-match fit exactly (Kendall's $\tau = 1.0$). Thus, the late entrant extends the leaderboard without perturbing it.

Counterfactual entrance cost. Playing all ten incumbents instead of 5 active-sample matches would have added five matches. Using the observed per-run averages ($1.12 per entrant solving run, $1.16 per incumbent solving run on an entrant task) and the unsampled incumbents' generation coverage (69 additional entrant solving runs, plus $5 \times 10 = 50$ additional incumbent solving runs on the entrant's bank), the extra matches would have cost roughly $136, so a full ten-pairing entrance is estimated at about $379. Thus, the active sampling procedure saved about a third of the entrance's evaluation spend, and an entrant's marginal cost scales with $\lceil \log_2 N \rceil + 1$ matches rather than the $O(N)$ pairings of a full round robin.

T TEXTUAL INTERFACE

The framework ships an interactive terminal interface for task generation and tournament management. Figure 14 shows the task-generation view, and Figure 15 shows the tournament view.

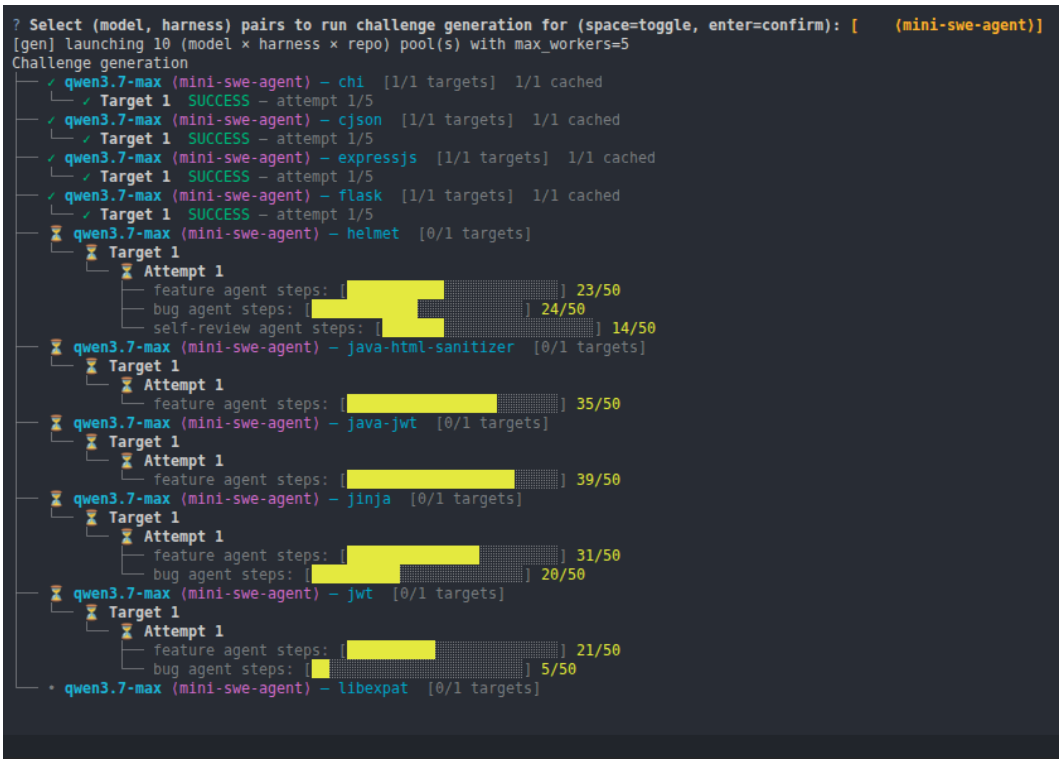


Figure 14: Textual interface for task generation. Rows correspond to numbered generation slots; each row tracks the feature/bug sessions, gate outcomes, and attempt history of that slot.

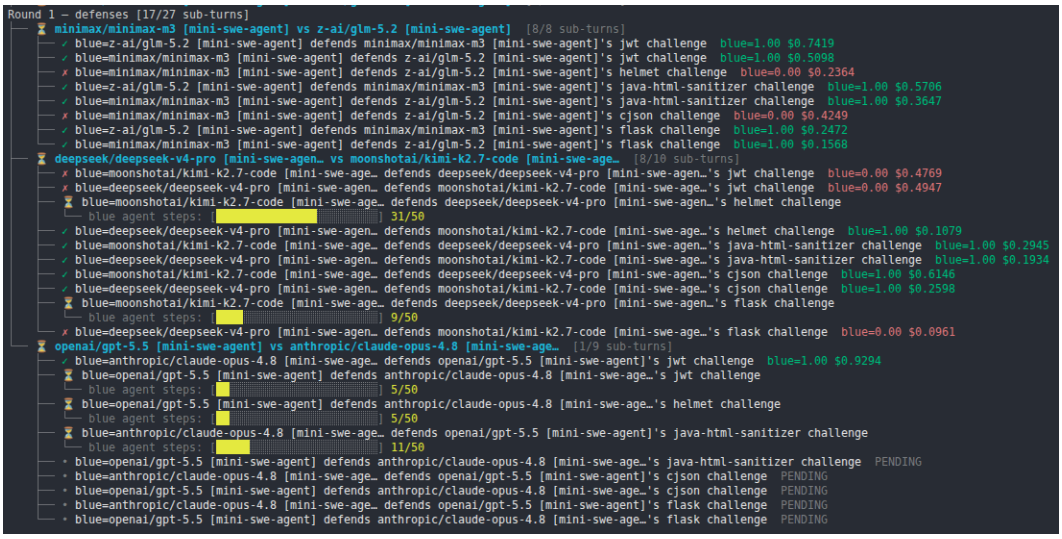


Figure 15: Textual interface for tournament scheduling. Pairings are drawn from the persisted schedule, solving runs are served from the cache, and completed matches report per-side outcomes and cost.