SWE-QA: BENCHMARKING CODING AGENTS FOR PROACTIVE RUNTIME QUALITY ASSURANCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly adopted in software engineering, demonstrating strong capabilities in resolving explicitly reported bugs in large-scale codebases. However, existing SWE benchmarks typically presuppose the availability of high-quality issue reports with detailed information, a premise that does not always hold in practice. Consequently, these benchmarks primarily assess code implementation capabilities, while offering limited evaluation of quality assurance abilities. To address this gap, we introduce SWE-QA, a benchmark for evaluating coding agents on proactive runtime quality assurance, comprising 123 tasks from 97 real-world repositories across 10 programming languages. Agents must analyze expected behavior, design and execute tests in the software's runtime environment, and produce evidence-grounded issue reports documenting discrepancies between expected and observed behavior. Extensive experiments reveal that discovering latent bugs triggered by unconventional boundary conditions, as well as identifying multiple bugs within a single task, remains highly challenging for frontier models. We believe SWE-QA provides a complementary perspective on SWE evaluation and that improved performance on it could help close the gap between autonomous bug discovery and resolution, bringing future coding agents closer to a fully autonomous, end-to-end software engineering workflow.

1 INTRODUCTION

Coding agents increasingly participate in software implementation: they translate specifications into code, extend existing systems, and repair reported defects. Issue-driven benchmarks such as SWE-bench and SWE-Bench Pro make the last capability measurable by providing a repository and an issue description, then checking the resulting patch (Jimenez et al., 2024; Deng et al., 2025). These tasks address an important part of software engineering, but the report itself is an input. In practice, identifying what is wrong, finding the conditions that expose it, and communicating an actionable diagnosis constitute substantial engineering work before a repair can begin.

We measure proactive runtime quality assurance, a capability complementary to implementation evaluation (Figure 1). Given a brief symptom, the agent investigates the relevant behavior, designs tests, explores execution trajectories, and connects an observed failure to the implementation that produces it. The deliverable is an issue report supported by reproducible evidence. Proactive means that the agent recovers the triggering conditions and the diagnosis from that symptom and from the software's runtime.

The central challenge is an evidence gap between an observable symptom and a useful issue. For example, a report that module-related command-line settings appear ineffective does not identify which state, sequence of operations, or implementation decision causes the problem. The agent must infer the intended behavior from available evidence, generate discriminating tests, and seek a runtime mismatch. A credible report connects four elements: a defensible expectation, a concrete trigger, an observed deviation, and a causal code location.

SWE-QA comprises 123 tasks from 97 repositories in 10 programming languages. Each task exposes a pinned buggy source snapshot and a concise, symptom-oriented case hint. A private behavioral contract defines the target discrepancy; a reference patch anchors eligible existing functions for deterministic localization. Evaluation collects reproduction evidence, checks the reported pinpoint,

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.

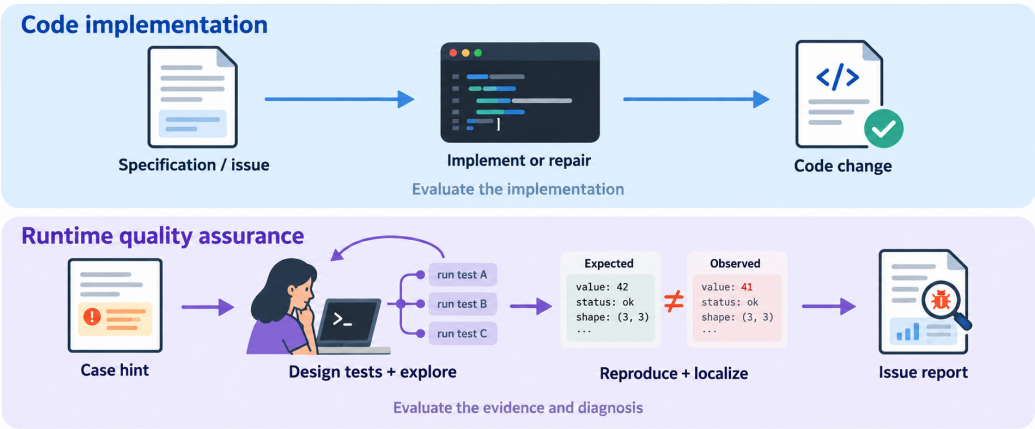

Figure 1: **Code implementation and runtime quality assurance are complementary.** Implementation starts from an explicit change request and produces code. SWE-QA starts from a case hint and, by testing the existing software, produces evidence and a diagnosis for a specified behavioral defect.

and applies a semantic report judge. A separate human-in-the-loop audit examines the task design itself for ambiguity, leakage, false positives, and false negatives. The audit score is a property of the task, and the benchmark score is a property of the agent.

Our contributions can be summarized as follows.

- We introduce SWE-QA, a benchmark for evaluating whether coding agents can turn brief symptom hints into reproducible bug reports through runtime investigation.
- We develop a construction and evaluation pipeline that uses human review to improve task quality and runtime evidence to assess issue reports.
- We conduct extensive experiments on frontier models and leading harnesses, highlighting the challenges of runtime QA and the role of agent design in successful investigation.

2 Related Work

Repository-level implementation and long-horizon tasks. SWE-bench established issue-driven repository evaluation, in which an agent receives a problem statement and produces a patch (Jimenez et al., 2024). SWE-Bench Pro extends this setting to more complex, enterprise-oriented tasks spanning larger changes and multiple files (Deng et al., 2025). More recent benchmarks challenge agents beyond short repair episodes. DeepSWE uses original engineering tasks and purpose-built functional verifiers (Huang et al., 2026); SWE-Marathon evaluates sustained software and technical workflows with multi-layer verification (Desai et al., 2026); FrontierSWE includes long-horizon implementation, optimization, scientific computing, and research tasks (Kondra et al., 2026). These benchmarks broaden the difficulty and scope of software engineering evaluation. Their objective remains completion of a requested change.

Proactive discovery and broader engineering workflows. Active-SWE directly challenges the assumption that a detailed issue is available, evaluating discovery and repair of bugs without report guidance (Li et al., 2026). It is closely related to our motivation, but differs in task scope and deliverable: SWE-QA provides a symptom to delimit the investigation and evaluates reproduction and diagnosis without asking for a repair. SWE Atlas broadens evaluation to codebase question answering, test writing, and refactoring, combining programmatic checks with task-specific rubrics (Raghavendra et al., 2026). These tasks also require exploration and can involve runtime reasoning. SWE-QA evaluates the investigation as a whole: infer the intended behavior, discover an unknown trigger through testing, connect the observed failure to its cause, and communicate evidence that another engineer can reproduce.

Agent interfaces and diagnostic evidence. ReAct interleaves reasoning with actions and environment observations (Yao et al., 2022); SWE-agent studies agent–computer interfaces for repository work (Yang et al., 2024), and OpenHands provides an extensible platform for software agents (Wang et al., 2024). These systems motivate controlling the harness when studying runtime investigation. Diagnostic evidence also has a longer history: delta debugging systematically isolates failure-inducing inputs (Zeller & Hildebrandt, 2002), while studies of human-facing bug reports examine what makes reports useful to developers (Bettenburg et al., 2008). Recent work studies report features and controlled report mutations for AI repair agents (Khatib et al., 2026). Our complementary focus is producing reproducible evidence and causal diagnosis from a symptom hint, then measuring their downstream value.

Evaluation reliability and model discrimination. Recent studies show why a task's stated objective and its verifier must be assessed together. DeepSWE motivates purpose-built verifiers rather than relying only on tests accompanying a historical fix (Huang et al., 2026). SWE-Bench Pro Verified studies solution leakage and inconsistencies between problem statements and evaluation tests (Zheng et al., 2026). Independently, SWE-Bench Pro V2 revises the public split and evaluation protocol, including instruction corrections, restricted network access, and re-grading on pristine environments (Dan et al., 2026). These efforts strengthen implementation evaluation; they also motivate human review of hint–contract alignment and false acceptance or rejection in SWE-QA. For comparing agents, Arena-Hard studies confidence-aware separability (Li et al., 2024). Section 4.3 reports a descriptive minimum-gap comparison on a fixed model set. LLM-as-a-judge research documents both human alignment and systematic judge biases (Zheng et al., 2023), motivating our separate measurements of human agreement and judge–human alignment.

3 SWE-QA

3.1 From a reported symptom to a diagnosed issue

SWE-QA evaluates the investigation that precedes a bug fix. Given a repository and a brief case hint, an agent must determine how the reported failure occurs, reproduce it in the software's runtime, and explain its cause in an evidence-grounded issue report. The score requires that explanation to connect the reported behavior to the implementation, so test design and runtime investigation are the substance of the task.

Task and interaction process. For task i, the public input $x_i = (S_i, E_i, h_i)$ contains a source snapshot, an execution environment, and a symptom hint. The private verifier input $z_i = (C_i, P_i)$ contains the behavioral contract and reference patch. Together with the agent's time budget B_i, these define the task $\mathcal{T}_i = (x_i, z_i, B_i)$. Given interaction history $H_{i,t}$, the agent–harness policy π selects a tool action $a_{i,t}$, and the runtime kernel $\mathcal{K}_i$ produces the next observation $o_{i,t+1}$:

$$a_{i,t} \sim \pi(\cdot \mid x_i, H_{i,t}), \qquad o_{i,t+1} \sim \mathcal{K}_i(\cdot \mid H_{i,t}, a_{i,t}). \tag{1}$$

Starting from $H_{i,0} = (o_{i,0})$, each action–observation pair is appended to the history. Actions include source inspection and test execution, and $\mathcal{K}_i$ carries persistent runtime state. At stopping index T_i, the trajectory is $\tau_i = H_{i,T_i}$, and the terminal policy returns $(R_i, A_i) \sim \pi_{\text{out}}(\cdot \mid x_i, \tau_i)$. The run stops at completion or when the time budget B_i is exhausted. That budget covers both investigation and submission.

The output is an issue report R_i together with reproduction artifacts A_i, in the form of an executable script, a written procedure, or both. The report states the discrepancy, the triggering conditions, the expected behavior, the causal location, and the root cause. The agent may inspect the source and interact with the runtime, and must leave the original source unchanged. Action selection and the report depend only on the public input x_i and the resulting observations. The behavioral contract and reference patch stay with the verifier, so the agent infers the intended behavior from the symptom and the software.

The case hint names an affected capability and an observable malfunction. Recovering the triggering sequence, the responsible function, and the repair is left to the investigation. For example, a report that module-related command-line settings appear ineffective requires the agent to determine which

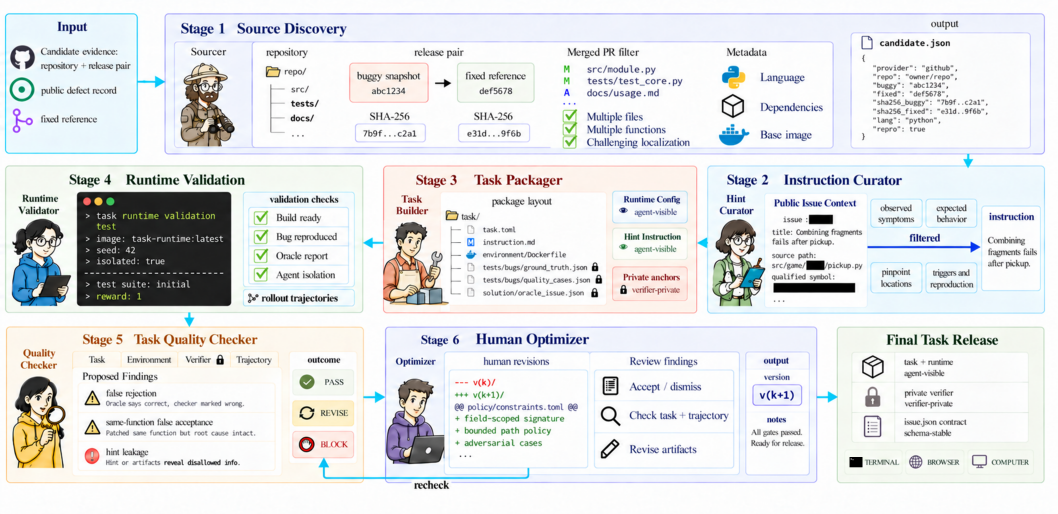

Figure 2: **From documented defects to runtime QA tasks.** Stages 1–3 turn repository evidence into an interactive task, separating the symptom an agent sees from the evidence used to judge its report. Stages 4–6 bring runtime trajectories into automated review and human revision, feeding corrections back into validation before the task is released.

states and operations expose the inconsistency. The agent develops hypotheses, designs discriminating tests, and revises the investigation as execution results arrive. Several trajectories may establish the same defect; the benchmark scores the evidence and the diagnosis.

3.2 Task construction pipeline

Figure 2 shows how SWE-QA turns documented defects into executable QA tasks through six stages. Starting from repository releases, defect reports, and merged fixes, the Sourcer identifies the buggy snapshot and fixed reference, establishing traceable evidence for each candidate. The Instruction Curator uses this evidence to define the target defect and derive a public symptom hint and a private behavioral contract. The hint leaves triggering sequences and causal locations for the agent to discover, while the contract records the expected behavior and conditions needed to assess its diagnosis. The Task Packager then combines the hint, investigation protocol, and buggy software into an interactive environment, keeping the contract, reference patch, and scoring rules on the private evaluation side. Once the candidate is executable, the Runtime Validator checks environment readiness, exercises the relevant behavior, and verifies that private evaluation assets remain inaccessible to the agent. Trial runs produce trajectories that reveal what an agent can observe and attempt, providing execution evidence for the Task Quality Checker to assess agreement among the hint, contract, and scoring rules. This review looks for answer leakage, acceptance of unrelated reports, and rejection of valid alternative reproductions, producing findings and a recommendation to pass, revise, or block the task. Human reviewers adjudicate these findings against source evidence and trajectories, revising the hint, environment, or verifier where needed. Revised candidates return to runtime validation and quality checking before approval, so problems observed during investigation feed back into task construction. The released task provides the agent with a symptom hint, runtime, and shared report format, paired with a private verifier for evaluating the resulting diagnosis. Appendix A provides implementation details.

3.3 Task statistics

SWE-QA contains 123 tasks drawn from 97 repositories across 10 programming languages. Figure 3 shows the task count for each language, using the language label in each task manifest. Python accounts for 25 tasks, followed by Rust (18), TypeScript (15), and Go (14). The collection spans developer tools, storage systems, services, numerical software, and games, so agents must adapt

their testing strategies to unfamiliar interfaces and execution models. Detailed counts and shares appear in Appendix C.

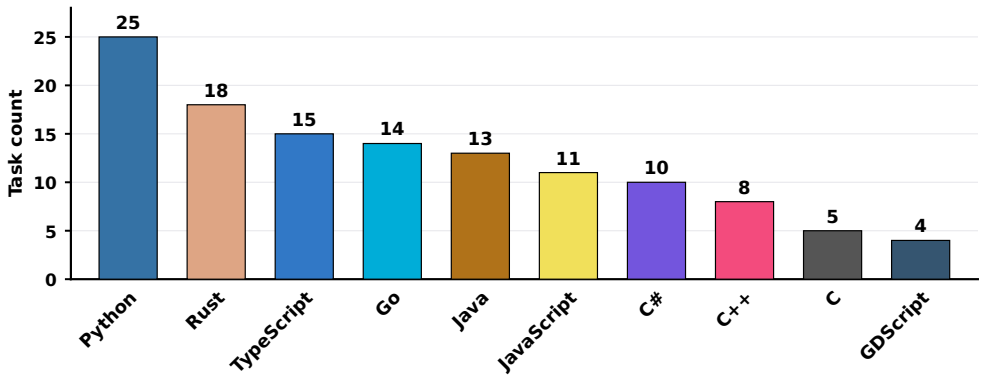

Figure 3: **Task distribution by programming language.** The 123 tasks cover 10 languages, ordered by task count. Each task is counted once using its manifest language.

3.4 Evaluating reproduction and diagnosis

An issue report must establish both the target failure and its cause. SWE-QA combines replay evidence from a fresh copy of the pinned software, deterministic matching against existing functions changed by the reference patch, and semantic assessment of the report. The semantic rubric assesses behavioral identity, reproduction evidence, and causal explanation, and it accepts alternative valid tests and semantically equivalent descriptions. Credit for localization requires a related failure and a supported explanation.

Verification functional and evaluation objective. Let $\varepsilon_i = \mathcal{R}_i(A_i)$ be the evidence record, including replay results, prose evidence, and any replay failure. The localization function $\mathcal{M}_i(R_i) \in \{0, 1\}$ uses S_i and P_i. When it returns 1, the semantic judge $\mathcal{J}_i$ uses the report, evidence, source, and private contract to assign a score in $[0, 1]$. Its rounded value is $Q_i = \mathrm{round}_2[\mathcal{J}_i(R_i, \varepsilon_i)]$, with acceptance threshold $\theta = 0.70$. Writing $r_i = \mathcal{V}_i(R_i, A_i)$ for the verification outcome, benchmark $\mathcal{D} = \{\mathcal{T}_i\}_{i=1}^{N}$ is scored by

$$r_i = \begin{cases} 1, & \mathcal{M}_i(R_i) = 1 \text{ and } Q_i \geq \theta, \\ 0, & \text{otherwise}, \end{cases} \qquad \widehat{\mathcal{S}}_{\mathcal{D}}(\pi) = \frac{100}{N} \sum_{i=1}^{N} r_i. \tag{2}$$

Here round_2 rounds to two decimal places. If localization fails, $r_i = 0$ and the judge is not called. When localization succeeds, the replay record, including a failed replay, is evidence for the judge. The reported score $\widehat{\mathcal{S}}_{\mathcal{D}}$ is the observed percentage of successes. Its expectation $\mathcal{S}_{\mathcal{D}}(\pi) = \mathbb{E}[\widehat{\mathcal{S}}_{\mathcal{D}}(\pi)]$ is the objective across repeated runs, averaging variability in the agent, the runtime, and the judge. Appendix A specifies the rubric and execution settings.

3.5 Human-in-the-loop quality refinement

Task quality depends on agreement between the public symptom, private behavioral contract, and verifier. An ambiguous hint can make a task underdetermined; an overly permissive verifier can accept an incorrect diagnosis; an overly narrow one can reject a valid reproduction. We use a Quality Checker Agent to identify these potential problems and propose revisions.

Each iteration includes human adjudication. A reviewer assesses the checker's suggestions, decides which are genuine task flaws, and revises the affected artifacts before the next audit. We track the mean task-quality score together with the mean numbers of checker-proposed and human-accepted flaws per task.

Figure 4 places both series on a shared iteration axis through V9.

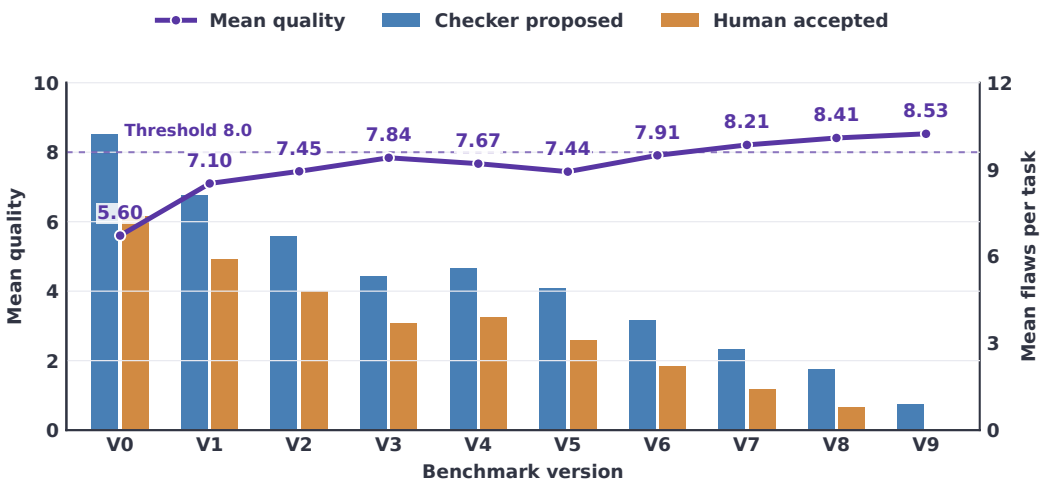

Figure 4: **Quality refinement across benchmark versions.** The line shows mean quality (left axis); grouped bars show checker-proposed and human-accepted flaws per task (right axis).

4 EXPERIMENTS

We organize the evaluation around four questions: how well agents complete runtime QA tasks, how the harness affects a fixed model, how the benchmark distinguishes agents, and how human and automated judgments align. The configuration catalog covers 21 models and seven harnesses; their primary reports, model documentation, and implementations are cited in Appendix B.1. Table 1 groups model results by harness and reports the best-scoring reasoning setting for each pair. Full configurations and resource profiles appear in the appendix. Section 4.4 reports the reliability results.

4.1 REASONING AND RUNTIME EXPLORATION

We evaluate the quality of the investigation. Additional reasoning helps when it produces a more discriminating test, exposes a hidden trigger, or corrects a faulty causal hypothesis. A long trace can also be an investigation that never narrows the problem.

The profiles show a broad trade-off between score and resource use. Flash configurations occupy a relatively efficient region, and GPT-6 Astra's highest-effort setting is non-monotonic. Appendix Figure 7 includes every configuration, including those omitted by best-per-pair selection.

4.2 HARNESS EFFECTS WITH THE MODEL HELD FIXED

Harness design affects how agents inspect a codebase, manage context, construct tests, and respond to failures. We compare Claude Code with alternative harnesses while holding the model and reasoning level fixed, then average over the levels shared by each pair. DeepSeek, Kimi, Qwen, and GLM are paired with DeepSeek CLI, Kimi CLI, Qwen Code, and mini-swe-agent, respectively. Score and output tokens are arithmetic means over the shared levels.

Claude Code scores higher in these paired comparisons. The comparison asks whether the harness helps the agent obtain and interpret execution evidence; token use is reported alongside the score. The pairing and aggregation rules are in Appendix B.

4.3 DISCRIMINATING BETWEEN AGENTS

A useful benchmark should distinguish even models whose scores are close. Let $\mathcal{M} = \{m_1, \ldots, m_K\}$ be a fixed set of $K \geq 2$ shared model identifiers and $s_m(B) \in [0, 1]$ the score under a specified evaluation configuration on benchmark B. We define the *minimum-gap score*

Harness	Model (effort)	Steps	Score (%)
Codex	GPT-6 Astra (xhigh)	113	**82.0**
	GPT-5.6 Sol (max)	93	79.5
	GPT-5.6 Terra (max)	103	75.8
	GPT-5.5 (xhigh)	82	73.6
	GPT-5.6 Luna (max)	102	71.4
Claude Code	Claude Opus 5 (max)	99	80.4
	Claude Sonnet 5 (max)	144	77.6
	DeepSeek V4.1 Flash (max)	89	76.4
	Claude Opus 4.8 (max)	120	73.2
	Kimi K3 (max)	98	72.1
	GLM-5.3 (max)	124	70.6
	Qwen3.8-Max (xhigh)	111	68.4
Gemini CLI	Gemini 3.8 Flash (high)	87	77.2
	Gemini 3.7 Flash (high)	112	67.1
	Gemini 3.1 Pro (high)	125	65.7
DeepSeek CLI	DeepSeek V4.1 Flash (max)	105	68.0
	DeepSeek V4 Pro (max)	177	58.7
	DeepSeek V4 Flash (max)	161	52.6
Kimi CLI	Kimi K3 (max)	104	71.0
	Kimi K2.6 (thinking)	159	56.4
Qwen Code	Qwen3.8-Max (xhigh)	124	62.3
	Qwen3.7-Plus (thinking)	150	49.5
mini-swe-agent	GLM-5.3 (max)	149	62.2
	GLM-5.3 Flash (max)	150	56.1
	GLM-5.2 (max)	199	45.8

Table 1: Scores grouped by harness. The best-scoring effort for each model–harness pair is shown in parentheses. Gray rows mark the highest score within each harness; the highest score overall is bold.

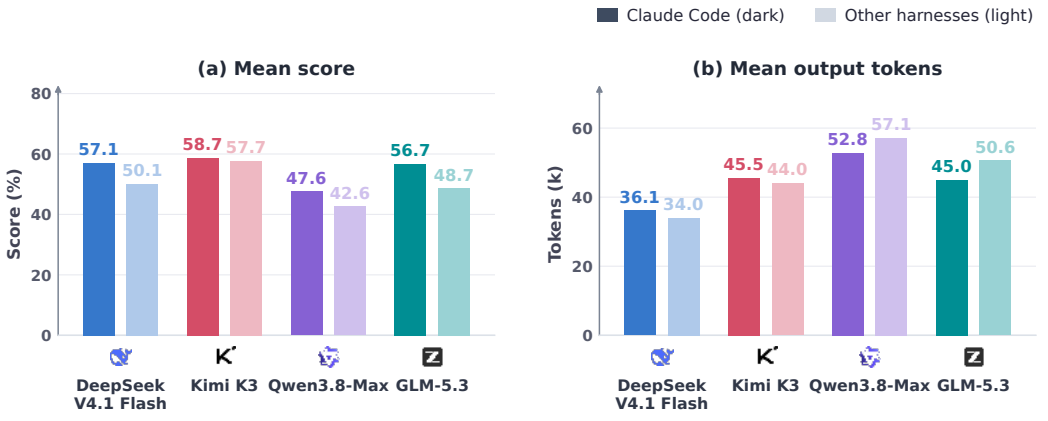

Figure 5: **Harness comparison at matched reasoning levels.** Mean score and output tokens for the same models in Claude Code and other harnesses.

separability as

$$\delta_{\mathcal{M}}(B) = \min_{1 \le i < j \le K} |s_{m_i}(B) - s_{m_j}(B)| = \min_{1 \le i < K} \big(s_{(i+1)}(B) - s_{(i)}(B)\big), \tag{3}$$

where $s_{(1)}(B) \le \cdots \le s_{(K)}(B)$ are the scores sorted separately on each benchmark, over the same model set. A larger $\delta_{\mathcal{M}}$ means the closest pair is farther apart.

Annotation Set	Count	Krippendorff's α [95% CI]
Reward= 0	124	0.8920 [0.8307, 0.9534]
Reward= 1	254	0.9180 [0.8718, 0.9641]
Overall Candidates	**378**	$\mathbf{0.9010}$ [$\mathbf{0.8619}$, $\mathbf{0.9399}$]

Table 2: Inter-annotator agreement on the semantic quality gate.

Model	Pearson r [95% CI]	p-value
Gemini 3.5 Flash	0.858 [0.8032, 0.8984]	< 0.0001
Claude 5 Opus	0.821 [0.7538, 0.8712]	< 0.0001
DeepSeek-V4 Flash	0.807 [0.7353, 0.8608]	< 0.0001
GPT-5.5	$\mathbf{0.903}$ [$\mathbf{0.8757}$, $\mathbf{0.9226}$]	< 0.0001

Table 3: Pearson correlation between semantic judges and human scores.

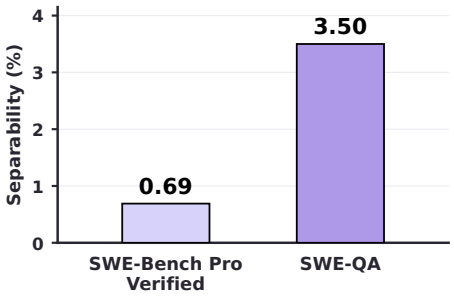

Figure 6: **Minimum-gap separability.** $\delta_{\mathcal{M}}$ (Eq. 3) is expressed in percent, as an absolute score difference across the same five models. Larger gaps indicate greater separation. Pro Verified scores are from Zheng et al. (2026).

Figure 6 compares this gap on five models shared with SWE-Bench Pro Verified (Zheng et al., 2026). The minimum gap is 3.50 percentage points on SWE-QA and 0.69 on Pro Verified. Harnesses differ across the two benchmarks, so the ratio describes those configurations. Model scores, closest pairs, and mean pairwise gaps are in Appendix B.3.

4.4 Reliability

Three annotators independently labeled 378 candidate reports from agent rollouts. Table 2 reports an overall Krippendorff's $\alpha = 0.901$ (Krippendorff, 2018), indicating consistent judgments on the semantic quality threshold. The annotation protocol is in Appendix B.2.

Table 3 reports the Pearson correlation (Pearson, 1895) between semantic judges and those human scores. GPT-5.5 (OpenAI, 2026b) has the highest correlation ($r = 0.903$) and is the judge used for the reported benchmark scores.

5 Conclusion

We introduced SWE-QA to evaluate the investigation that turns a brief symptom into a reproducible bug report. Through tasks drawn from real repositories, human-reviewed construction, and evaluation grounded in runtime evidence, the benchmark makes reproduction and causal diagnosis explicit targets of coding-agent evaluation. Our experiments show that QA performance varies across models and harnesses, and that greater reasoning effort does not consistently improve results. Matched comparisons further demonstrate the role of harness design when the underlying model is held fixed. These findings motivate studying how agents gather and use execution evidence throughout an investigation. By making the investigation preceding a fix measurable, SWE-QA addresses an evaluation gap left by benchmarks that assume a detailed issue report is already available. Progress on this benchmark could help close the gap between autonomous bug discovery and resolution, bringing coding agents closer to a fully autonomous, end-to-end software engineering workflow.

References

Alibaba Cloud. Alibaba Cloud Model Studio: Recommended models. Model documentation, 2026. URL https://www.alibabacloud.com/help/en/model-studio/models. Accessed September 26, 2026.

Anthropic. Claude Code. Software repository, 2026a. URL https://github.com/anthropics/claude-code. Accessed September 26, 2026.

Anthropic. Claude Opus 4.8 system card. System card, 2026b. URL https://anthropic.com/claude-opus-4-8-system-card. Accessed September 26, 2026.

Anthropic. Claude Opus 5 system card. System card, 2026c. URL https://anthropic.com/claude-opus-5-system-card. Accessed September 26, 2026.

Anthropic. Claude Sonnet 5 system card. System card, 2026d. URL https://anthropic.com/claude-sonnet-5-system-card. Accessed September 26, 2026.

Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? (revision 1.1). Technical report, Saarland University, 2008. URL https://www.st.cs.uni-saarland.de/publications/details/bettenburg-tr-2008/.

Soham Dan, Jeff Da, Sanyam Bhutani, Miguel Romero Calvo, Ying Liu, and Daniel Yue Zhang. SWE-Bench Pro V2: A cleaner, harder-to-game leaderboard. *Scale Labs Blog*, September 2026. URL https://labs.scale.com/blog/swe-bench-pro-v2.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. Technical report, 2026a. URL https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/resolve/main/DeepSeek_V41_Tech_Report.pdf.

DeepSeek-AI. DeepSeek Harness. Software repository, 2026b. URL https://github.com/deepseek-ai/deepseek-harness. Accessed September 26, 2026.

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. arXiv:2606.19348, 2026. URL https://arxiv.org/abs/2606.19348.

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? arXiv:2509.16941, 2025. URL https://arxiv.org/abs/2509.16941.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Daniel Wang, Derek Chen, Pranav Raja, Albert Liu, Marek Šuppa, Nevasini Sasikumar, Luyang Kong, Erik Quintanilla, Xiangyi Li, Ivan Bercovich, and Steven Dillmann. SWE-Marathon: Can agents autonomously complete ultra-long-horizon software work? Technical report, 2026. URL https://www.swe-marathon.org/swe-marathon-paper.pdf.

GLM-5-Team et al. GLM-5: From vibe coding to agentic engineering. arXiv:2602.15763, 2026. URL https://arxiv.org/abs/2602.15763.

Google. Gemini CLI. Software repository, 2026. URL https://github.com/google-gemini/gemini-cli. Accessed September 26, 2026.

Google DeepMind. Gemini 3.1 Pro model card. Model card, 2026a. URL https://deepmind.google/models/model-cards/gemini-3-1-pro/. Accessed September 26, 2026.

Google DeepMind. Gemini 3.5 Flash model card. Model card, 2026b. URL https://deepmind.google/models/model-cards/gemini-3-5-flash/. Accessed September 26, 2026.

Google DeepMind. Gemini 3.7 Flash model card. Model card, 2026c. URL https://deepmind.google/models/model-cards/gemini-3-7-flash/. Accessed September 26, 2026.

Google DeepMind. Gemini 3.8 Flash model card. Model card, 2026d. URL https://deepmind.google/models/model-cards/gemini-3-8-flash/. Accessed September 26, 2026.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. arXiv:2607.07946, 2026. URL https://arxiv.org/abs/2607.07946.

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

Lara Khatib, Noble Saji Mathews, Meiyappan Nagappan, Pengyu Nie, and Thomas Zimmermann. What makes a good bug report for an AI agent? arXiv:2607.07593, 2026. URL https://arxiv.org/abs/2607.07593.

Kimi Team et al. Kimi K3: Open frontier intelligence. arXiv:2607.24653, 2026. URL https://arxiv.org/abs/2607.24653.

Rishyanth Kondra, Sanket Mhatre, Akshit Kumar, Evan Chu, Bilal Bakht Ahmad, Ayush Nangia, Rajan Agarwal, Arpan Dasgupta, Animesh Sinha, Bhuvanesh Sridharan, Krupa Dave, Brendan Graham, Guanyu Song, Anirudh Rahul, Wei Hern Lim, Abishek Thangamuthu, Ramneet Singh, Danna Liu, Navid Pour, Calvin Chen, and Justus Mattern. FrontierSWE v2. *Proximal Blog*, 2026. URL https://www.frontierswe.com/blog/v2.

Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, 4 edition, 2018.

Haobin Li, Ping Deng, Weizhong Qian, Liang Jiang, Zhenyu Huang, Mouxing Yang, and Xi Peng. Active-SWE: Benchmarking coding agents for proactive bug fixing without issue reports. arXiv:2608.04682, 2026. URL https://arxiv.org/abs/2608.04682.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. From crowdsourced data to high-quality benchmarks: Arena-Hard and Bench-Builder pipeline. arXiv:2406.11939, 2024. URL https://arxiv.org/abs/2406.11939.

Moonshot AI. Kimi Code CLI. Software repository, 2026a. URL https://github.com/MoonshotAI/kimi-code. Accessed September 26, 2026.

Moonshot AI. Kimi-K2.6 model card. Model card, 2026b. URL https://huggingface.co/moonshotai/Kimi-K2.6. Accessed September 26, 2026.

OpenAI. Codex CLI. Software repository, 2026a. URL https://github.com/openai/codex. Accessed September 26, 2026.

OpenAI. GPT-5.5 system card. System card, 2026b. URL https://openai.com/index/gpt-5-5-system-card/. Accessed September 26, 2026.

OpenAI. GPT-5.6 system card. System card, 2026c. URL https://deploymentsafety.openai.com/gpt-5-6. Accessed September 26, 2026.

OpenAI. GPT-6 Astra system card. System card, 2026d. URL https://deploymentsafety.openai.com/gpt-6-astra. Accessed September 26, 2026.

Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

Qwen Team. Qwen Code. Software repository, 2026. URL https://github.com/QwenLM/qwen-code. Accessed September 26, 2026.

Mohit Raghavendra, Soham Dan, Miguel Romero Calvo, Yannis Yiming He, Johannes Baptist Mols, Gautam Anand, Cole McCollum, Edgar Arakelyan, Vijay Bharadwaj, Andrew Park, Jeff Da, MohammadHossein Rezaei, Bing Liu, Brad Kenstler, and Yunzhong He. SWE Atlas: Benchmarking coding agents beyond issue resolution. arXiv:2605.08366, 2026. URL https://arxiv.org/abs/2605.08366.

SWE-agent Team. mini-swe-agent. Software repository, 2026. URL https://github.com/SWE-agent/mini-swe-agent. Accessed September 26, 2026.

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

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. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. arXiv:2210.03629, 2022. URL https://arxiv.org/abs/2210.03629.

Z.ai. GLM-5.2: Overview. Model documentation, 2026a. URL https://docs.z.ai/guides/llm/glm-5.2. Accessed September 26, 2026.

Z.ai. GLM-5.3: Overview. Model documentation, 2026b. URL https://docs.z.ai/guides/llm/glm-5.3. Accessed September 26, 2026.

Z.ai. GLM-5.3-Flash/FlashX: Overview. Model documentation, 2026c. URL https://docs.z.ai/guides/vlm/glm-5.3-flash. Accessed September 26, 2026.

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. doi: 10.1109/32.988498. URL https://www.st.cs.uni-saarland.de/publications/details/zeller-tse-2002/.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and chatbot arena. arXiv:2306.05685, 2023. URL https://arxiv.org/abs/2306.05685.

Pujun Zheng, Zixin Shang, Shufan Jiang, Wenhui Tian, Dongsheng Zhu, Zerun Ma, Dingbo Yuan, and Qi Zhang. SWE-Bench Pro Verified: A reliable benchmark for software engineering agents. arXiv:2609.08149v2, 2026. URL https://arxiv.org/abs/2609.08149v2.

A IMPLEMENTATION AND REPRODUCIBILITY DETAILS

Instruction and package details. Task-specific hints average 12.04 whitespace-separated words (range 7–22); full instructions average 488.07 words. Each current task package contains nine files: an instruction, manifest, agent Dockerfile, and six verifier files. No solution directory is packaged. The agent may consult public technical documentation and install dependencies, but must not inspect Git history or modify the original source. It writes one report to `/logs/artifacts/report.md`, accompanied by `reproduce.sh` and/or `reproduce.md`. Unconfirmed report fields should state "Not sure."

Task generation. The export includes SHA-256 hashes for shared instruction rendering, quality and reward templates, the quality-checker prompt, and the reported evaluation data. Deterministic regeneration, not per-task manual editing, is the maintenance mechanism. The construction pipeline's evidence collection and consistency checks should not be confused with end-to-end runtime certification of every task.

Budgets and versioning. Current task manifests configure a 7,200-second agent budget and a 3,600-second verifier budget. The packaged reward implementation and shared reward template use a 1,200-second replay timeout and a 1,800-second judge timeout. The separate verifier Dockerfile pins Codex CLI 0.147.0 and RewardKit 0.1.7, with `gpt-5.5` and `xhigh` reasoning for the judge. These do not specify the versions of the evaluated agent harnesses. A release should additionally pin source hashes, image digests, API model snapshots, package dependencies, and all harness configurations.

Execution order. The entry-point launches the reward program, which prepares runtime evidence before deterministic pinpoint matching and semantic judging. These stages are not parallel evaluations. The semantic judge is not reached when the required localization gate fails. A replay failure is otherwise passed as evidence, not an independent zero-reward gate. Errors in preparing the verifier environment must be reported separately from an agent's inability to reproduce the case.

Replay boundary. The verifier uses a copy of the pinned buggy source under `/sandbox/software/`, not a fixed revision. It runs the reproduction script with reduced privileges, a sanitized environment, and read-only submitted artifacts, recording command output, return status, trajectory, and source-change checks in `runtime_evidence.json`. Source fingerprints can reveal changes to pre-existing files, but do not constitute a complete tamper-proof record of transient modifications. Script return codes and logs remain agent-influenced evidence. A prose trajectory is useful to the judge but lacks the assurance of an independent executable replay. The benchmark should report the proportions of scripted, prose-only, failed, and missing evidence separately.

Report schema. The required report headings are Description, Triggering Conditions, Expected Behavior, Reproduction, Pinpoint, and Root Cause. Pinpoint columns are File, Function, Lines, and Rationale. Paths are repository-relative POSIX paths; line references identify the buggy snapshot. Reproduction may refer to the artifact evidence or state "Not sure." Failure to reproduce should be reported honestly. A detailed issue report is the scored deliverable, not a patch.

Quality rubric. The five dimensions and weights are behavioral identity and expected outcome (0.25), reproduction evidence (0.25), causal localization (0.25), root-cause/reproduction coherence (0.15), and scope/consistency (0.10). Missing or invalid reproduction evidence caps quality at 0.20; missing/generic expected behavior at 0.20; a different bug, opposite outcome, or material contradiction at 0.25; a missing behavior-changing condition at 0.69; no resolvable function at 0.20; a non-causal function at 0.49; caller-only or nearby localization at 0.60; a missing or source-contradicting explanation at 0.69; and undifferentiated unrelated candidates at 0.60. With raw dimension scores s_{ik} and applicable cap c_i, the judge is instructed to return $q_i = \min(\sum_k w_k s_{ik}, c_i)$. The implementation compares $Q_i = \mathrm{round}(q_i, 2)$ with $\theta = 0.70$. Caps are prompt-level instructions, not separately enforced numerical checks. Deterministic matching requires a repository-relative file, an existing function modified or deleted by the reference patch, and overlapping reported lines; wholly added functions do not qualify.

Human adjudication ledger. For each task and iteration, record the version and immutable source/template hashes, checker score, a distinct flaw identifier, category, evidence, human decision and rationale, and the version in which an accepted flaw is resolved. Score records should exist even for tasks with zero proposed flaws. Compute per-task averages over all audited tasks, not only those with reported issues. The supplied CSV schema separates task scores from flaw-level decisions, avoiding duplicate counting when a task has several suggestions. For N_v audited tasks, the three aggregates are $\bar{Q}_v = N_v^{-1} \sum_i Q_{iv}$, $\bar{F}_v = N_v^{-1} \sum_i F_{iv}$, and $\bar{A}_v = N_v^{-1} \sum_i A_{iv}$, with $0 \leq A_{iv} \leq F_{iv}$. Accepted suggestions and resolved flaws are distinct quantities.

B Experimental settings

Configuration selection. The reported evaluation covers 25 model–harness pairs and 98 configurations. The main table selects the highest score within each pair, breaking ties by lower cost, and then groups rows by harness. Effort is displayed next to the model; the globally highest score is bold. A complete configuration should identify the provider endpoint, immutable model snapshot, harness version, reasoning setting, source image, and execution budget.

Matched-effort comparisons. Harness comparisons use the intersection of reasoning levels for each model: four each for DeepSeek V4.1 Flash and Qwen3.8-Max, and three each for Kimi K3 and GLM-5.3. The reported score and output tokens are equal-weight arithmetic means across these levels. Cross-benchmark comparison instead uses the highest listed effort under a fixed harness policy: native CLI where listed, and mini-swe-agent for GLM. Dated model variants with no exact catalog match are excluded. This selection differs from the best-per-pair main table.

B.1 Model and harness sources

Table 4 maps every evaluated model and harness to a primary source. Shared reports are cited once per model family where they cover the listed variants. Model cards and official documentation identify variants without a separate technical report. The GLM-5 technical report provides family-level background (GLM-5-Team et al., 2026); the table cites documentation specific to each evaluated GLM variant. GPT-5.5, Claude Opus 5, and DeepSeek V4 Flash appear in both agent and judge evaluations; Gemini 3.5 Flash is used only in the judge comparison. Software and documentation references were checked on September 26, 2026; these sources identify the systems, while run-specific versions belong in the configuration records.

Model or harness	Source type	Reference
Models evaluated as QA agents		
GPT-6 Astra	System card	(OpenAI, 2026d)
GPT-5.6 Sol, Terra, Luna	System card	(OpenAI, 2026c)
GPT-5.5	System card	(OpenAI, 2026b)
Claude Opus 5	System card	(Anthropic, 2026c)
Claude Sonnet 5	System card	(Anthropic, 2026d)
Claude Opus 4.8	System card	(Anthropic, 2026b)
Gemini 3.8 Flash	Model card	(Google DeepMind, 2026d)
Gemini 3.7 Flash	Model card	(Google DeepMind, 2026c)
Gemini 3.1 Pro	Model card	(Google DeepMind, 2026a)
DeepSeek V4 Pro, Flash	Technical report	(DeepSeek-AI et al., 2026)
DeepSeek V4.1 Flash	Technical report	(DeepSeek-AI, 2026a)
Kimi K3	Technical report	(Kimi Team et al., 2026)
Kimi K2.6	Model card	(Moonshot AI, 2026b)
Qwen3.8-Max, Qwen3.7-Plus	Model documentation	(Alibaba Cloud, 2026)
GLM-5.3	Model documentation	(Z.ai, 2026b)
GLM-5.2	Model documentation	(Z.ai, 2026a)
GLM-5.3 Flash	Model documentation	(Z.ai, 2026c)

Continued on next page

Table 4 continued

Model or harness	Source type	Reference
Additional model in the judge comparison		
Gemini 3.5 Flash	Model card	(Google DeepMind, 2026b)
Evaluated harnesses		
Codex CLI	Software repository	(OpenAI, 2026a)
Claude Code	Software repository	(Anthropic, 2026a)
Gemini CLI	Software repository	(Google, 2026)
DeepSeek CLI (DeepSeek Harness)	Software repository	(DeepSeek-AI, 2026b)
Kimi CLI (Kimi Code CLI)	Software repository	(Moonshot AI, 2026a)
Qwen Code	Software repository	(Qwen Team, 2026)
mini-swe-agent	Software repository	(SWE-agent Team, 2026)

Table 4: **Primary sources for evaluated models and harnesses.** Source types distinguish technical reports from system/model cards, model documentation, and software repositories.

For mini-swe-agent, we additionally cite the SWE-agent paper recommended by its official repository (Yang et al., 2024).

B.2 Reliability annotation

Each of the 378 candidates pairs an agent rollout with its extracted issue report. Three annotators labeled whether the report passed the semantic quality threshold $\theta = 0.70$. Reward$= 0$ denotes a score below 0.70 (124 candidates) and Reward$= 1$ a score at or above 0.70 (254 candidates). These labels cover the semantic gate. The benchmark reward in Equation 2 also requires reference-anchored localization. Annotator agreement is nominal Krippendorff's α (Krippendorff, 2018). Alignment between a judge and the human scores is Pearson's r (Pearson, 1895).

B.3 Selected-model score separability

Equation 3 measures the smallest absolute gap among all $\binom{K}{2}$ pairs in a fixed model set. Every difference between nonadjacent sorted scores is a sum of nonnegative adjacent gaps, so its minimum is attained by an adjacent pair. This proves the equivalence of the two expressions, including ties. The metric is invariant to model-list permutations and satisfies $0 \le \delta_{\mathcal{M}}(B) \le 1/(K-1)$. Adding models can decrease the minimum; comparisons must therefore retain the same model set. Treating individual models as singleton pools recovers the adjacent-pool formulation when the pool order agrees with the score order. We impose no common ranking across benchmarks and do not transfer saturation thresholds from tier-level screening.

To summarize all pairs in addition to the closest pair, we report the mean absolute pairwise gap:

$$D_{\mathcal{M}}(B) = \frac{2}{K(K-1)} \sum_{1 \le i < j \le K} |s_{m_i}(B) - s_{m_j}(B)|. \tag{4}$$

Table 5 records all inputs and summary statistics. The sorted adjacent gaps are $(8.89, 0.69, 2.46, 0.96)$ percentage points for Pro Verified and $(12.90, 3.50, 8.80, 8.50)$ for SWE-QA. Thus the minimum-gap ratio is $3.50/0.69 = 5.07$, while the mean-pairwise-gap ratio is $15.94/5.83 = 2.73$. These calculations use the reported score precision without rounding intermediate gaps.

Statistical separation and validation. For model score intervals I_a and I_b, confidence-aware separability can be summarized as $\binom{K}{2}^{-1} \sum_{a<b} \mathbb{1}[I_a \cap I_b = \varnothing]$. Neither $\delta_{\mathcal{M}}$ nor $D_{\mathcal{M}}$ estimates this statistic. Paired resampling of task outcomes, clustered by repository, can account for shared tasks and related codebases; each resample must recompute the minimum over all pairs rather than hold the observed closest pair fixed. Repeated runs are needed to assess additional sampling variability. Judge calibration should include human-adjudicated valid and invalid reports, repeated judgments, and false-accept/false-reject rates. Replaying on fixed software is a proposed negative control, not

Model / statistic	**Pro Verified**	**SWE-QA**
GPT-5.6 Sol	61.97	79.50
Kimi K3	62.93	71.00
GLM-5.3	58.82	62.20
DeepSeek V4 Pro	49.93	58.70
GLM-5.2	59.51	45.80
Minimum pairwise gap $100\delta_{\mathcal{M}}$	0.69	**3.50**
Mean pairwise gap $100D_{\mathcal{M}}$	5.83	**15.94**
Score range	13.00	33.70

Table 5: **Inputs and summaries for the fixed five-model comparison.** Model scores are percentages; all gap statistics are percentage points. Pro Verified scores come from the published comparison (Zheng et al., 2026).

part of the current scoring implementation. The task score does not measure multi-bug recall. A single binary outcome over 123 tasks changes the score by approximately 0.813 percentage points.

C Repository and language coverage

Language	Tasks	Share (%)
Python	25	20.3
Rust	18	14.6
TypeScript	15	12.2
Go	14	11.4
Java	13	10.6
JavaScript	11	8.9
C#	10	8.1
C++	8	6.5
C	5	4.1
GDScript	4	3.3
Total	123	100.0

Language labels follow the task manifests rather than a repository-wide source-code census. Repositories may be multilingual. Repository coverage counts distinct repository identifiers; it does not imply that all modules of each repository are evaluated.

D Efficiency profiles

E Complete configuration snapshot

The following table lists every model–harness–effort combination, including non-best points. Rows are grouped by harness, with reasoning effort in parentheses and provider icons preceding model names, as in Table 1. These identifiers are configuration labels, not independently verified API capability claims.

Harness	Model (effort)	Steps	Score (%)
Codex	GPT-6 Astra (low)	52	60.8
	GPT-6 Astra (medium)	76	70.9
	GPT-6 Astra (high)	95	77.7
	GPT-6 Astra (xhigh)	113	**82.0**
	GPT-6 Astra (max)	130	80.0
	GPT-5.6 Sol (none)	37	29.6
	GPT-5.6 Sol (low)	51	49.8

Continued on next page

Table 6 continued

Harness	Model (effort)	Steps	Score (%)
	GPT-5.6 Sol (medium)	63	62.4
	GPT-5.6 Sol (high)	73	73.6
	GPT-5.6 Sol (xhigh)	83	78.3
	GPT-5.6 Sol (max)	93	79.5
	GPT-5.6 Terra (none)	41	25.8
	GPT-5.6 Terra (low)	57	44.9
	GPT-5.6 Terra (medium)	70	56.3
	GPT-5.6 Terra (high)	81	67.8
	GPT-5.6 Terra (xhigh)	92	72.6
	GPT-5.6 Terra (max)	103	75.8
	GPT-5.5 (none)	33	23.7
	GPT-5.5 (low)	48	41.3
	GPT-5.5 (medium)	60	54.8
	GPT-5.5 (high)	71	65.9
	GPT-5.5 (xhigh)	82	73.6
	GPT-5.6 Luna (none)	41	19.4
	GPT-5.6 Luna (low)	56	34.6
	GPT-5.6 Luna (medium)	69	47.8
	GPT-5.6 Luna (high)	80	59.7
	GPT-5.6 Luna (xhigh)	91	66.8
	GPT-5.6 Luna (max)	102	71.4
Claude Code	Claude Opus 5 (low)	40	53.8
	Claude Opus 5 (medium)	58	64.6
	Claude Opus 5 (high)	73	73.3
	Claude Opus 5 (xhigh)	86	78.5
	Claude Opus 5 (max)	99	80.4
	Claude Sonnet 5 (low)	58	47.6
	Claude Sonnet 5 (medium)	84	58.5
	Claude Sonnet 5 (high)	106	68.9
	Claude Sonnet 5 (xhigh)	125	74.9
	Claude Sonnet 5 (max)	144	77.6
	DeepSeek V4.1 Flash (off)	36	27.8
	DeepSeek V4.1 Flash (low)	57	53.6
	DeepSeek V4.1 Flash (high)	73	70.6
	DeepSeek V4.1 Flash (max)	89	76.4
	Claude Opus 4.8 (low)	48	40.8
	Claude Opus 4.8 (medium)	70	52.6
	Claude Opus 4.8 (high)	88	62.8
	Claude Opus 4.8 (xhigh)	104	70.1
	Claude Opus 4.8 (max)	120	73.2
	Kimi K3 (low)	39	43.4
	Kimi K3 (high)	72	60.7
	Kimi K3 (max)	98	72.1
	GLM-5.3 (low)	50	40.7
	GLM-5.3 (high)	91	58.9
	GLM-5.3 (max)	124	70.6
	Qwen3.8-Max (off)	44	22.4
	Qwen3.8-Max (low)	71	41.7
	Qwen3.8-Max (medium)	92	57.9
	Qwen3.8-Max (xhigh)	111	68.4
Gemini CLI	Gemini 3.8 Flash (low)	35	52.1
	Gemini 3.8 Flash (medium)	64	69.2
	Gemini 3.8 Flash (high)	87	77.2
	Gemini 3.7 Flash (low)	45	38.4
	Gemini 3.7 Flash (medium)	82	53.9
	Gemini 3.7 Flash (high)	112	67.1
	Gemini 3.1 Pro (low)	50	36.2
	Gemini 3.1 Pro (medium)	92	52.6
	Gemini 3.1 Pro (high)	125	65.7

Continued on next page

Table 6 continued

Harness	Model (effort)	Steps	Score (%)
DeepSeek CLI	DeepSeek V4.1 Flash (off)	39	22.4
	DeepSeek V4.1 Flash (low)	64	46.1
	DeepSeek V4.1 Flash (high)	84	64.0
	DeepSeek V4.1 Flash (max)	105	68.0
	DeepSeek V4 Pro (off)	66	14.8
	DeepSeek V4 Pro (low)	108	31.8
	DeepSeek V4 Pro (high)	143	49.0
	DeepSeek V4 Pro (max)	177	58.7
	DeepSeek V4 Flash (off)	60	12.2
	DeepSeek V4 Flash (low)	97	25.3
	DeepSeek V4 Flash (high)	130	41.7
	DeepSeek V4 Flash (max)	161	52.6
Kimi CLI	Kimi K3 (low)	38	42.7
	Kimi K3 (high)	74	59.5
	Kimi K3 (max)	104	71.0
	Kimi K2.6 (off)	59	18.0
	Kimi K2.6 (thinking)	159	56.4
Qwen Code	Qwen3.8-Max (off)	46	18.6
	Qwen3.8-Max (low)	75	36.3
	Qwen3.8-Max (medium)	100	53.2
	Qwen3.8-Max (xhigh)	124	62.3
	Qwen3.7-Plus (off)	56	13.0
	Qwen3.7-Plus (thinking)	150	49.5
mini-swe-agent	GLM-5.3 (low)	56	33.8
	GLM-5.3 (high)	106	50.2
	GLM-5.3 (max)	149	62.2
	GLM-5.3 Flash (low)	56	27.0
	GLM-5.3 Flash (high)	106	42.0
	GLM-5.3 Flash (max)	150	56.1
	GLM-5.2 (none)	74	12.2
	GLM-5.2 (high)	141	30.7
	GLM-5.2 (max)	199	45.8

Table 6: Complete configurations grouped by harness. Effort appears in parentheses; the highest score is bold.

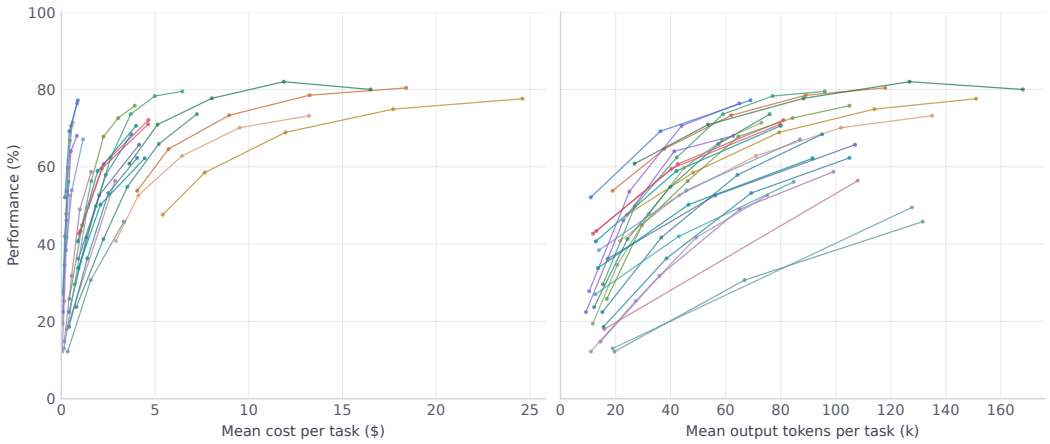

Figure 7: **Configuration-level results.** Each connected line is one model–harness pair ordered by declared effort. No monotonicity adjustment is applied.

F AI USE STATEMENT

Generative AI assisted manuscript restructuring, language editing, implementation inspection, LaTeX and plotting code, and the visual rendering of Figures 1 and 2. Numerical inputs and deterministic plotting scripts are retained separately from the generated graphics. Figure 5 is rendered deterministically from the frozen numerical inputs. The construction pipeline uses an LLM to draft task artifacts; another judges submitted reports, while a Quality Checker Agent assists human task review. The authors remain responsible for reviewing the final manuscript and its supporting evidence.