Position: Coding Benchmarks Are Misaligned with Agentic Software Engineering

Maria I. Gorinova
Tessl
London, UK
maria@tessl.io

Macey Baker
Tessl
London, UK
macey@tessl.io

Amy Heineike
Tessl
London, UK
amy@tessl.io

Maksim Shaposhnikov
Tessl
London, UK
max@tessl.io

Rob Willoughby
Tessl
London, UK
robw@tessl.io

Dru Knox
Tessl
London, UK
dru@tessl.io

Abstract

Coding agents have become a major mode of software engineering, but the benchmarks we use to compare them were designed in a pre-agent era: they collapse model, harness, and environment into a single end-to-end score, typically computed against one reference solution, with no component-level signal for iteration. **We argue that current coding benchmarks are misaligned with agentic software engineering.** A coding agent in practice is not a model: it is a *system harness* — a composite of models, harnesses, contexts, environments, and feedback signals, any one of which can move the benchmark score by margins comparable to those between adjacent model generations. We discuss three symptoms: (i) benchmark scores conflate the model with the rest of the harness; (ii) grading against a single reference solution penalises equally valid alternatives; and (iii) the absence of signal at the level of individual harness components makes the end-to-end system score difficult to iterate on.

1 Introduction

Coding agents [3, 10, 31, 45, 49] are now a major mode of software engineering. They open and merge pull requests, write internal libraries, and increasingly take on multi-day engineering work under human supervision. They are composite systems, consisting of a large language model (LLM) in a tool-use loop, scaffolding, environment, and context. Each of these components can shift the end-to-end benchmark score by margins comparable to those between adjacent model generations [1, 29].

But the benchmarks we use to compare them were designed for an earlier object of study: the ability of an LLM to generate working code in one go. SWE-Bench [20], HumanEval [9], MBPP [5], LiveCodeBench [19], and BigCodeBench [53] all share the same structure: a single model, a single harness, and a single environment together produce a single number — an end-to-end system score with no signal at the level of individual components — which is often compared against a single reference solution.

The choice of benchmark is not neutral: it implicitly shapes how methods are judged and which research directions get pursued, even if the benchmark only partially captures the unit of interest [11].

We take the position that current coding benchmarks are misaligned with agentic software engineering: they grade only a small part of what we build, against constructs we do not want. Closing the gap requires benchmarks designed around the structure of agentic systems, rather than around individual reference solutions. Such benchmarks would treat the agent as the composite system it is, expose signal at the level of individual components, and ground correctness in independent behavioural specifications rather than in any single reference solution. The hardest open problem inside this programme is *operationalisation*: specifying what we want the system to do in terms that can be measured, without encoding how the agent should attempt it.

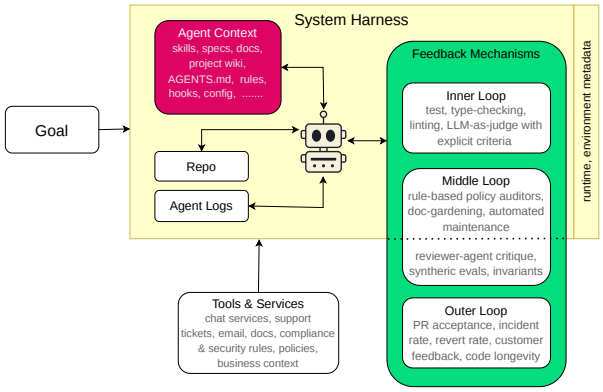

Figure 1: The system harness around a coding agent. Yellow: components the harness modifies or produces. Outside: what it reads but does not control. Feedback signals split into inner-, middle-, and outer-loop tiers; each tier also splits into agent-controlled (the harness can rewrite the check) vs. external (PR comments, production outcomes).

2 The System Harnesses

A coding agent is not a model: it is part of a *system harness*, an orchestration layer around one or more LLMs that manages tasks, environments, and feedback over time. We distinguish two levels of this orchestration. The *agent harness* is a language model interacting with tools, working towards a single task, with some system prompt and context to draw on. Most artefacts described as "coding agents" are agent harnesses in this sense — Claude Code [3], Codex [31], Cursor Agent [10], SWE-Agent [49], OpenHands [45], and many others. The *system harness* is the outer orchestration: it transforms

higher-level goals into concrete tasks, dispatches each to one or more agent harnesses, manages the environment they act on, and routes their outputs through feedback that approximates whether the work is acceptable. Practical agentic coding at scale operates at the level of the system harness [4, 33]; current coding benchmarks operate at the level of the agent harness. Recent examples include Symphony [21] and GasCity [14]. We also built and open-sourced NS2,[1] an issue-driven system harness running a four-tool agent loop under a stack of deterministic and agent-arbitrated checks. In NS2, GitHub issues are the coordination primitive: a product-manager agent decomposes a feature issue into vertical-slice child issues; software-engineering agents implement those issues in isolated sessions; reviewer, test-quality, smoke-testing, and PR-building agents consume issue and pull-request events to decide whether work should move forward, be revised, or be described for review. Building and operating NS2 surfaced many of the misalignments this paper articulates, and we draw on it as a concrete reference throughout. Many harnesses, including NS2, treat the issue tracker as the durable state machine for work and spawn per-task agent sessions.

The system harness has five recurring components (Figure 1): (i) *tasks*, units of work derived from higher-level goals; (ii) one or more *agent harnesses*, configurable executors composed of model, prompt, tools, and loop, which the system harness may tune or treat as black boxes; (iii) the *environment* the repository and runtime under change, together with integrated external services (issue tracker, CI, deployment surface); (iv) *context*, a curated projection of the environment and of harness-authored material — skills, plugins, hooks, specs — loaded into a particular invocation; and (v) *feedback signals*: anything the harness reads to refine a solution or to refine itself, including tests, types, linters, formal verification, LLM-as-judge rubrics, PR comments, reviewer critique, production incidents, and longer-horizon business signals. Within feedback, *verifiers* are the strict subset that return a pass/fail verdict suitable for blocking — tests, type-checks, linters, and binary rubrics; the broader feedback category includes qualitative review and outcome signals that inform rather than gate.

We categorise feedback into three tiers by scope, latency and trust. *Inner-loop* signals (seconds to minutes: tests, types, lint, compile) are fast and cheap but narrow. They operate inside a single change attempt, usually at commit or pull-request scope, and give the agent immediate steer on whether the current edit is executable, well-typed, policy-compliant, or worth further iteration. *Middle-loop* signals (minutes to hours: reviewer requests, simulation, maintenance agents, score rubrics) aggregate over a wider slice of work. A *maintenance agent* is a scheduled or on-demand agent that inspects a project, its PRs, or its agent traces, then files or implements recurring quality improvements to the software project. Middle-loop signals capture properties that are too broad or too judgement-dependent for a unit test: recurring review criticism, drift from project conventions, duplicated abstractions, agent inefficiency visible in logs, or before/after health metrics for a maintenance run. *Outer-loop* signals (days to weeks: PR acceptance, revert rate, incident reports, customer feedback) are closest to ground truth but delayed and confounded; they measure whether shipped work survived human review, production, and use, often without clean attribution to one prompt, task, or change. A productive system harness uses all three: inner signals to refine a solution in-loop, middle signals to surface recurring issues and quality drift, and outer signals to calibrate which inner and middle proxies are worth trusting. Middle-loop signals are therefore useful when they predict or improve outer-loop outcomes such as fewer reverts, fewer review rounds, or a lower human-intervention rate. A second, orthogonal axis distinguishes signals the harness can modify (e.g. tests) from signals it cannot (human PR comments, business outcomes). All signals can take part in the harness's *self-improvement* loop, in which accumulated logs and recurring failures feed back into the harness's own components [22, 33]. NS2 illustrates the pattern: max-pedantic lint, a coverage threshold, and dependency-graph unit tests act as inner-loop strict verifiers; mutation testing, LCOM cohesion checks, and daily agentic architecture and test-quality reviews are middle-loop feedback; friction reports from a smoke-testing agent, post-merge revert signals, and production incidents are outer-loop feedback. The agent writes and maintains the lint rules and rubrics that constrain it.

3 Related Work

We read existing coding-agent evaluation work through the lens of the system harness §2.

Coding Benchmarks. Coding benchmarks fall into two families. The first scores models on short, self-contained problems with hidden tests: HumanEval [9], MBPP [5], LiveCodeBench [19] (with contamination control), and BigCodeBench [53]. These were designed when the artefact under test was a model, and they correctly target the model component of the agent harness. They are not designed to discriminate between system harnesses. The second family grades agents on patches to real repositories. SWE-Bench [20] requires an agent's patch to make a held-out `FAIL_TO_PASS` set pass while keeping a `PASS_TO_PASS` set passing; both test sets are derived from the original pull request, a construction we return to in §4.2. The benchmark has since iterated to address distinct shortcomings: Verified [30] curates 500 human-validated tasks, Multimodal [50] adds visual domains, and Pro [37] broadens the task horizon and language coverage with human-validated tasks (and is now recommended by OpenAI in place of Verified [34]). SWE-rebench [6] trades human validation for scale, generating 32k+ tasks across 20 languages. Beyond the issue-fix shape, Terminal-Bench [28] broadens evaluation to terminal tasks (typically 1–20 minutes) and has become a de-facto frontier reporting standard; Frontier-SWE [35] targets ultra-long-horizon performance and ML-research challenges. Adjacent suites include SWE-Lancer [32], RE-Bench [48], MLE-bench [8], τ-bench [52], AgentBench [27], and the Aider polyglot [15]. These benchmarks vary the task domain, time horizon, and verifier shape, but share the same set-up: the agent is paired with a fixed environment and verifier, and a single end-to-end pass rate is reported. In the language of §2, the rest of the system harness is folded into the protocol rather than treated as part of the artefact under test.

Validity and the Benchmark–deployment Gap. Recent papers expose validity problems in the SWE-Bench-style set-up: SWE-Bench+ [2] documents solution leakage in issue text and weak-test

[1] https://github.com/drufball/ns2

passes; Liang et al. [26] report file-localisation behaviour consistent with memorisation; Wang et al. [46] use differential testing to show 7.8% of resolved patches fail developer-written tests and 29.6% diverge from the gold patch's behaviour; and Whitfill et al. [47] find that many resolved patches would not be merged under ordinary maintainer review. Li et al. [23, 24] measure the gap at the other end: across 456k agent-authored PRs in 61k repositories, real-world acceptance rates are 35–64% — well below the >70% headline figures on Verified. Their AIDev dataset is useful beyond the gap it documents. Because it is drawn from live repositories rather than a curated benchmark, it can report acceptance, review turnaround, and code complexity in place of a single pass rate, which is closer to the measurement we argue for. A separate line of work has begun to treat the harness itself as the object of measurement. Fan et al. [12] report that the same scaffold varies 3–7× in token-budget effectiveness across base LLMs and conclude that effectiveness is a property of the scaffold–model integration, not of either component alone. SkillsBench [25] measures the lift attributable to agent skills, and Meta-Harness [22] treats the harness as an object of optimisation, searching the harness-code space with an outer proposer. Our position is consistent with this trajectory: if the harness is the artefact, the harness is what should be measured.

Agentic Software Engineering. The framing of an agentic software engineering (SE) discipline is taking shape in parallel. Hassan et al. [17] call for an "SE 3.0" research roadmap around structured human–agent collaboration, with *merge-readiness packs* replacing test-pass-as-success — "passing tests alone is no longer enough". Our argument is the measurement counterpart: if the artefact is a composite system, the benchmark must score the composite system. We draw on work that treats evaluation as a measurement problem. Wallach et al. [44] argue that evaluating GenAI is a social-science measurement challenge, distinguishing the *construct* (e.g. "solves the bug") from the *operationalisation* (*how* it is measured); Jacobs and Wallach [18] formalise the chain of validity a measurement must satisfy to be informative about its construct. We use this language directly in §4: single-reference anchoring (§4.2) is a *content-validity* claim; bundle conflation (§4.1) is a *discriminant-validity* claim — a benchmark that cannot separate model from harness is measuring something, but not the thing it labels.

4 Three Symptoms of the Misalignment

We discuss three symptoms of the misalignment between current benchmarks and agentic coding systems.

4.1 Conflating the Model with the Harness

This conflation is not new: Dehghani et al. [11] made a closely related point about non-agent benchmarks, and the SWE-Bench community has rediscovered it incrementally since. Our position is that it has not been sufficiently acted on; the cost of inaction has grown because the model is a small part of what gets used in practice, and the remedy likely needs structural change at the benchmark level, alongside individual care.

Table 1 shows success rates for a single fixed model (Claude Opus 4.6) across several agent harnesses on Terminal-Bench, a benchmark of terminal-based tasks in containerised environments. Each task gives the agent an English instruction, a sandboxed environment, and hidden tests; solving it requires ordinary terminal work such as inspecting files, running commands, debugging failures, editing code or configuration, and recovering from intermediate mistakes [28]. On this fixed task distribution, differences of 20 percentage points or more appear across harnesses. Practitioner reports document 4–10 point swings for Claude Opus 4.5 between standardised and custom scaffolds on SWE-Bench Verified [29], and the OpenHands harness reaches 77.6% with comparable models that score several points lower under the standardised mini-SWE-Agent harness [45]. The effect is not just additive: Fan et al. [12] report that effectiveness "is not an inherent property of the scaffold" — it emerges from how the scaffold integrates with the base model, with model swaps moving resolve rate 2–3× at fixed scaffold. Across more than 200,000 SWE-Bench runs, AI21 [1] find that orchestration choices, container allocations, and evaluation seeds materially move the pass rate at fixed model and fixed harness; Anthropic report similar infrastructure-level noise inside Anthropic's own evaluation pipeline [39].

Table 1: Entries from the TerminalBench [28] leaderboard showing success rates for Claude Opus 4.6 across agent harnesses on a fixed task distribution. Within a fixed task distribution, success rates can vary by 20 percentage points or more — a range comparable to differences between model generations.

Rank	Agent	Model	Agent Org	Accuracy (%)
4	ForgeCode	Opus 4.6	ForgeCode	79.8 ± 1.6
8	Capy	Opus 4.6	Capy	75.3 ± 2.4
11	Terminus-KIRA	Opus 4.6	KRAFTON AI	74.7 ± 2.6
14	TongAgents	Opus 4.6	Bigai	71.9 ± 2.7
17	Droid	Opus 4.6	Factory	69.9 ± 2.5
20	Crux	Opus 4.6	Roam	66.9 (N/A)
22	Mux	Opus 4.6	Coder	66.5 ± 2.5
28	Terminus 2	Opus 4.6	Terminal-Bench	62.9 ± 2.7
40	Claude Code	Opus 4.6	Anthropic	58.0 ± 2.9

The model is fixed across the rows, so the spread cannot be explained as a difference in model capability. It is showing that the agent harness — prompt, tool interface, action loop, environment handling, retry behaviour, and conventions for using the terminal — is part of the measured object. A model may also have been trained or tuned under particular tool-use conventions, so a harness can be well or poorly matched to the model even before any task-specific reasoning begins. Another under-specified variable is inference effort: some model APIs let the caller directly change the amount of inference compute used to produce an answer, trading thoroughness against speed and cost, including across tool calls. Changing this setting can therefore move the quality of the output even when the model name and agent code are nominally unchanged.

But recent work often publishes single-harness, single-number comparisons regardless. The result is attributed to the model, while it is a property of the agent harness and environment as a whole; the model is one component among several. A leaderboard entry of the form "Model M, 65% on SWE-Bench Verified" is uninformative

about whether M would resolve a given test under different scaffolding or environment; comparing two such numbers is comparing two systems, not two models. End-to-end numbers are informative, but we claim they are *under-specified*, and that the unit being measured is not the unit being used in practice.

Suggested remedy. The fix is structural rather than methodological. Leaderboard maintainers and benchmark stewards should require relevant metadata at submission: what model, agent harness version, environment hash, and dataset version were used. Additionally, submissions should include at least one ablation across a non-model axis against a fixed baseline.

4.2 Anchoring on a Single Reference Solution

Many coding benchmarks grade a solution by its closeness to a single reference. SWE-Bench is the canonical instance for agentic work [20]: the `FAIL_TO_PASS` and `PASS_TO_PASS` test sets are derived from the test files modified in the original pull request, encoding a particular decomposition — which functions exist, their signatures, etc. An agent that resolves a flaky test by restating the API at a different level of abstraction is judged not on whether the bug is fixed, but on whether the reference tests still hold. In measurement terms, the patch is a proxy for the construct [18, 44].

Such grading is fair only when we specify tasks tightly enough that the agent has no real choice but to make the same implementation decisions as the reference solution. In practice, work is rarely this tight, and it is rarely just bug-fixing: developers ask agents to *define* the API, upgrade dependencies, evolve abstractions, or choose between architectural shapes. Practitioners consistently identify *spec quality* rather than model capability as the primary bottleneck [38, 42]. SWE-Bench instances, by contrast, are selected for tractability: each comes with a clearly filed issue and a cleanly merged patch. The benchmark is therefore doubly anchored: the output is graded against the reference patch, and the input is preselected to the standard of a well-formed issue.

This construction also embeds known weaknesses. Aleithan et al. [2] report 32.67% solution leakage in issue text and 31.08% passes under insufficient tests; Wang et al. [46] show via differential testing that 7.8% of resolved patches fail developer-written tests and 29.6% diverge from the gold patch's runtime behaviour; Liang et al. [26] document file-localisation consistent with memorisation.

A deeper issue is that single-reference grading mistakes both the construct and the grain. The reference encodes one solution among many: we want agents that can refactor, restructure, and pick among reasonable alternatives. And, in narrow domains such as compiler optimisation or kernel autotuning, find shapes no reference patch encodes. The shortcomings of single-reference grading are well established in machine learning more broadly [43].

The hidden unit tests also grade only local behaviour. They cannot see what distinguishes good code from working code: choice of abstractions, architectural fit, system design. An agent can pass every test while degrading the codebase in ways a reviewer would reject on sight. The methodological move is to grade not on closeness to a particular solution but on a broader definition of functional correctness and on design-level quality (code is reused rather than duplicated, new abstractions follow project conventions, the dependency graph stays sound). These are *invariants* — conditions that should hold across many candidate solutions, and across many PRs in the same codebase. Recent practitioner work moves in this direction: skill-adherence evaluations [40] grade against separately-authored policies; abstraction-adherence checks [41] verify structural properties without prescribing an implementation; ProgramBench [51] grades via agent-generated behavioural tests rather than source-code comparison. All three decouple the verifier from any particular candidate solution. The remaining work is articulation: specifying invariants as rubrics that can be graded reliably, and choosing tasks for which those invariants apply. We claim this is the central open problem in agentic-coding evaluation — specifying *what* we want the system to do in terms that do not encode *how*.

Suggested remedy. Replace single-reference-derived test sets with multi-shape behavioural verifiers — property tests, reference oracles, or differential tests against alternative implementations. Where a single gold patch is retained, declare which behaviours are required and which are incidental to the reference implementation.

4.3 The Absence of Component-Level Signal

End-to-end agent runs on a single benchmark task can take hours [35], yet each task yields only a small amount of signal. As we saw in §2, a modern agentic system — e.g. the one described by OpenAI [33], or NS2 with its stack of linters, dependency-graph unit tests, mutation testing, agentic reviewers, and a smoke-testing agent — has many components, each affecting the overall result. If one component is failing, the evaluation of an end-to-end task will capture the failure, but we will not necessarily be able to tell which component is faulty. To determine how to improve the overall harness, we might need to resort to running ablation experiments, thus making the improvement loop even more time-consuming.

Practitioners building toward autonomous systems describe a continuous improvement cycle in which failures must be diagnosed and fixed at the component level: context, tooling, verifier, or task decomposition [33, 38]. An end-to-end score shows that something failed; it does not say what to fix. Without component-level signal, this cycle degrades to intuition-guided ablation. If the harness is a composition of components, we should aim to evaluate components separately. This is the same logic that splits software testing into unit and integration tests [7]: an integration test reflects deployment more faithfully but does not say *which* component broke.

Recent evaluation work moves in this direction on the task side: Ribeiro et al. [36] decompose tasks into capabilities with unit-style assertions; Dehghani et al. [11] show that aggregated scores conceal which tasks drive the ranking. But the system under test is still a black box. The same decomposition should apply to the system itself: each component evaluated in isolation, as well as in composition.

Component-level evaluation for agentic systems is sparse, especially for coding. LLM-only benchmarks (e.g. one-shot code generation) effectively evaluate the model component, and a few techniques evaluate skills as a stand-alone context [25, 40]. Recent work in adjacent literature begins to target individual components: PEEK [16] scores agents on long-context aggregation and in-context learning rather than end-to-end completion, treating orientation knowledge as an evaluable artefact; DecisionBench [13] evaluates how well an agent delegates sub-tasks across a pool of models.

Neither targets coding directly, but both illustrate the shape: per-component verifiers that hold the rest of the harness constant. Some harness components are themselves evaluation targets for others: in NS2, mutation testing evaluates the quality of a unit-test suite, and an agentic linter-quality review evaluates the lint configuration. The harness components form a stack of verifiers-of-verifiers; reporting only the end-to-end pass rate flattens this stack.

Suggested remedy. Treat the components of Fig. 1 as evaluation targets in their own right, answering questions such as "How effective is the context in aiding the agent?", "How well does the agent follow agreed invariants?", and "How effective is the agent in converting policy to deterministic verifiers?". For maintenance agents, report before/after deltas on scoped health metrics — complexity, duplication, dead code, dependency cycles, or churn hotspots — rather than only whether the final PR merged.

5 Alternative Views

"End-to-end scores reflect real usage." We agree. We are arguing against using *only* end-to-end metrics and against treating the resulting score as a property of the model rather than of the harness. In software engineering, the same question was settled in favour of having *both* unit and integration tests.

"Decomposed evaluations are too costly." The dominant cost in current practice is the opportunity cost of misattributing improvements and selecting systems based on misleading signals. Even partial decomposition is helpful: adding one component-level metric alongside the end-to-end score already improves the signal.

"A reference solution is a reasonable gold standard." It is, when the task is specified tightly enough that the reference is the only reasonable shape. But the ambition of agentic software engineering is broader than producing passing patches: we want to evaluate design, abstraction choice, and architectural fit — qualities single reference patches do not encode and hidden test cannot see.

6 Call to Action

We call on the community to act on the three remedies in §4: report harness-aware metadata, move from single-reference test sets to verifiers that admit multiple valid solution shapes, and develop methods for component-level evaluation alongside end-to-end scores. Each is tractable but non-trivial. And underneath all three sits a harder problem: how do we state what we want a coding system to do in terms an automated grader can apply, without prescribing how? This is the operationalisation gap of Wallach et al. [44] applied to agentic coding, and it is the decisive constraint on the next generation of benchmarks. Until it closes, benchmarks will keep grading agents on how closely they resemble the solution that closed an issue — whereas the ambition is to let them do better.

References

[1] AI21. 2025. Scaling Agentic Evaluation: Lessons from 200,000 SWE-bench Runs. https://www.ai21.com/blog/scaling-agentic-evaluation-swe-bench/.

[2] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. SWE-Bench+: Enhanced Coding Benchmark for LLMs. (2024). arXiv:2410.06992 https://arxiv.org/abs/2410.06992.

[3] Anthropic. 2025. Claude Code. https://claude.com/product/claude-code.

[4] Anthropic. 2025. Effective Harnesses for Long-Running Agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. (2021). arXiv:2108.07732 [cs.PL] https://arxiv.org/abs/2108.07732.

[6] Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. 2026. SWE-rebench V2: Language-Agnostic SWE Task Collection at Scale. arXiv:2602.23866 [cs.SE] https://arxiv.org/abs/2602.23866.

[7] Kent Beck. 2002. *Test-driven development: by example.* Addison-Wesley Professional.

[8] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. 2024. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. (2024). arXiv:2410.07095 https://arxiv.org/abs/2410.07095.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374.

[10] Cursor. 2025. Cursor Agents. https://cursor.com/agents.

[11] Mostafa Dehghani, Yi Tay, Alexey A Gritsenko, Zhe Zhao, Neil Houlsby, Fernando Diaz, Donald Metzler, and Oriol Vinyals. 2021. The benchmark lottery. *arXiv preprint arXiv:2107.07002* (2021).

[12] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. 2025. SWE-Effi: Re-Evaluating Software AI Agent System Effectiveness Under Resource Constraints. (2025). arXiv:2509.09853 https://arxiv.org/abs/2509.09853.

[13] Yuxuan Gao, Megan Wang, Yi Ling Yu, Zijian Carl Ma, and Ao Qu. 2026. DecisionBench: A Benchmark for Emergent Delegation in Long-Horizon Agentic Workflows. (2026). arXiv:2605.19099 https://arxiv.org/abs/2605.19099.

[14] Gastown Hall. 2026. GasCity. https://github.com/gastownhall/gascity.

[15] Paul Gauthier. 2024. The Aider Polyglot Coding Benchmark. https://aider.chat/2024/12/21/polyglot.html.

[16] Zhuohan Gu, Qizheng Zhang, Omar Khattab, and Samuel Madden. 2026. PEEK: Context Map as an Orientation Cache for Long-Context LLM Agents. (2026). arXiv:2605.19932 https://arxiv.org/abs/2605.19932.

[17] Ahmed E Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa, and Dong Qiu. 2025. Agentic software engineering: Foundational pillars and a research roadmap. *arXiv preprint arXiv:2509.06216* (2025).

[18] Abigail Z. Jacobs and Hanna Wallach. 2021. Measurement and Fairness. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT).* 375–385. doi:10.1145/3442188.3445901 https://arxiv.org/abs/1912.05511.

[19] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination-Free Evaluation of Large Language Models for Code. In *International Conference on Learning Representations (ICLR).* arXiv:2403.07974 https://arxiv.org/abs/2403.07974.

[20] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations (ICLR).* arXiv:2310.06770 https://arxiv.org/abs/2310.06770.

[21] Alex Kotliarskyi, Victor Zhu, and Zach Brock. 2026. An open-source spec for Codex orchestration: Symphony. https://openai.com/index/open-source-codex-orchestration-symphony/.

[22] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-Harness: End-to-End Optimization of Model Harnesses. (2026). arXiv:2603.28052 https://arxiv.org/abs/2603.28052.

[23] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE 3.0): How Autonomous Coding Agents Are Reshaping Software Engineering. (2025). arXiv:2507.15003 https://arxiv.org/abs/2507.15003.

[24] Hao Li, Haoxiang Zhang, and Ahmed E Hassan. 2026. AIDev: Studying AI coding agents on GitHub. *arXiv preprint arXiv:2602.09185* (2026). https://arxiv.org/abs/2602.09185.

[25] Xiangyi Li, Wenbo Chen, Yimin Liu, et al. 2026. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. (2026). arXiv:2602.12670 https://arxiv.org/abs/2602.12670.

[26] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. *arXiv preprint arXiv:2506.12286* (2025). arXiv:2506.12286 https://arxiv.org/abs/2506.12286.

[27] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2024. AgentBench: Evaluating LLMs as Agents. In *International Conference on Learning Representations (ICLR).* arXiv:2308.03688 https://arxiv.org/abs/2308.03688.

[28] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, et al. 2026. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. (2026). arXiv:2601.11868 https://arxiv.org/abs/2601.11868.

[29] Morph Labs. 2025. SWE-Bench Pro: A Detailed Analysis of Scaffold-Driven Score Variance. https://www.morphllm.com/swe-bench-pro.

[30] OpenAI. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/.

[31] OpenAI. 2025. Introducing Codex. https://openai.com/index/introducing-codex/.

[32] OpenAI. 2025. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering? (2025). arXiv:2502.12115 https://arxiv.org/abs/2502.12115.

[33] OpenAI. 2026. Harness Engineering. https://openai.com/index/harness-engineering/.

[34] OpenAI. 2026. Why SWE-bench Verified No Longer Measures Frontier Coding Capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/.

[35] Proximal Labs. 2026. Frontier-SWE: A Benchmark of Long-Horizon Software Engineering Tasks. https://www.frontierswe.com/blog.

[36] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th annual meeting of the association for computational linguistics*. 4902–4912.

[37] Scale AI. 2025. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? arXiv:2509.16941 https://arxiv.org/abs/2509.16941.

[38] Brian Scanlan. 2026. How we use Claude Code today at Intercom. https://www.linkedin.com/pulse/how-we-use-claude-code-today-intercom-brian-scanlan-eb7cc/.

[39] Gian Segato and Engineering at Anthropic. 2026. Quantifying infrastructure noise in agentic coding evals. https://www.anthropic.com/engineering/infrastructure-noise.

[40] Maksim Shaposhnikov. 2025. A Proposed Framework For Evaluating Skills. *Tessl Blog* (2025). https://tessl.io/blog/a-proposed-framework-for-evaluating-skills-research-eng-blog/.

[41] Maksim Shaposhnikov, Maria I. Gorinova, Rob Willoughby, and Dru Knox. 2025. A Proposed Evaluation Framework for Coding Agents: Tiles Enhance Proper Use of Public APIs by 35%. *Tessl Blog* (2025). https://tessl.io/blog/proposed-evaluation-framework-for-coding-agents/.

[42] StrongDM. 2025. StrongDM Software Factory. https://factory.strongdm.ai/. Field notes on non-interactive agentic development.

[43] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

[44] Hanna Wallach, Meera Desai, A. Feder Cooper, Angelina Wang, Chad Atalla, Solon Barocas, Su Lin Blodgett, Alexandra Chouldechova, Emily Corvi, P. Alex Dow, et al. 2025. Position: Evaluating Generative AI Systems Is a Social Science Measurement Challenge. In *International Conference on Machine Learning (ICML)*. arXiv:2502.00561 https://arxiv.org/abs/2502.00561.

[45] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations (ICLR)*. arXiv:2407.16741 https://arxiv.org/abs/2407.16741.

[46] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. (2025). arXiv:2503.15223 https://arxiv.org/abs/2503.15223.

[47] Parker Whitfill, Cheryl Wu, Joel Becker, and Nate Rush. 2026. Many SWE-bench-Passing PRs Would Not Be Merged into Main. https://metr.org/notes/2026-03-10-many-swe-bench-passing-prs-would-not-be-merged-into-main/.

[48] Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua Clymer, Jai Dhyani, et al. 2025. RE-Bench: Evaluating Frontier AI R&D Capabilities of Language Model Agents Against Human Experts. In *International Conference on Machine Learning (ICML)*. arXiv:2411.15114 https://arxiv.org/abs/2411.15114.

[49] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*. arXiv:2405.15793 https://arxiv.org/abs/2405.15793.

[50] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *International Conference on Learning Representations (ICLR)*. arXiv:2410.03859 https://arxiv.org/abs/2410.03859.

[51] John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. 2026. ProgramBench: Can Language Models Rebuild Programs From Scratch? arXiv:2605.03546 [cs.SE] https://arxiv.org/abs/2605.03546

[52] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. (2024). arXiv:2406.12045 https://arxiv.org/abs/2406.12045.

[53] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *International Conference on Learning Representations (ICLR)*. arXiv:2406.15877 https://arxiv.org/abs/2406.15877.