Design Principles for Rigorous AI4SE Benchmarking

MALIHEH IZADI, Google, Switzerland
ERIC BODDEN, Universität Paderborn, Germany
MARCEL BÖHME, CISPA Helmholtz Center for Information Security, Germany
PREMKUMAR T. DEVANBU, University of California - Davis, United States
MEHDI KESHANI, Universität Zürich, Switzerland
CLAIRE LE GOUES, Carnegie Mellon University, United States
PETROS MANIATIS, Google, United States
PEDRO ORVALHO, Artificial Intelligence Research Institute, CSIC, Spain
PATRICK RONDON, Google, United States
SATISH CHANDRA*, Meta, United States
MICHAEL PRADEL, CISPA Helmholtz Center for Information Security, Germany

Artificial intelligence for software engineering has become one of the most consequential application areas for large language models, where benchmark improvements shape public opinion. But software engineering tasks are often ambiguous; repository mining leaks information in subtle ways; and test suites are only partial correctness specifications. How well, then, do benchmark improvements really reflect practical progress? We distill a set of principles for designing stronger software engineering benchmarks across five dimensions: task definition, data gathering, data validation, success measurement, and benchmark maintenance. These principles emphasize data integrity, transparent validation, meaningful baselines, and statistically rigorous evaluation. By adopting such practices, the community can turn benchmarks from static leaderboard targets into reliable scientific infrastructure for measuring and accelerating genuine progress.

Additional Key Words and Phrases: Benchmarking, Evaluations, Software Engineering, AI4SE, LLMs, Agents

ACM Reference Format:
Maliheh Izadi, Eric Bodden, Marcel Böhme, Premkumar T. Devanbu, Mehdi Keshani, Claire Le Goues, Petros Maniatis, Pedro Orvalho, Patrick Rondon, Satish Chandra, and Michael Pradel. 2026. Design Principles for Rigorous AI4SE Benchmarking. 1, 1 (October 2026), 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software engineering is emerging as the highest-impact near-term application of AI and LLMs. Every day, we hear of the latest frontier model scoring ever more on popular benchmarks like SWE-Bench [5], now at more than 80%. Hence, there is a temptation to conclude: *"Software engineering is 80% solved!"* However, most software engineers will not agree. The gap between 80% on a benchmark and 80% in practice is likely much wider than it appears.

Benchmarks have been phenomenally useful to machine learning research progress, giving model developers a stable yardstick to optimize against and enabling apples-to-apples comparisons across approaches. Benchmark performance clearly has market implications: Cognition Labs, for example, came into the limelight because of the significant jump in the scores their AI agent, *Devin*,

*This work was done in the author's personal capacity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/10-ART
https://doi.org/XXXXXXX.XXXXXXX

achieved on SWE-Bench.[1] Perceived competence on the basis of benchmarks can even influence public policy! The US government's *AI Action Plan* was influenced by results from benchmarks in software engineering and cybersecurity that suggested rapid progress in this area.[2] However, there remains a clear gap between what popular AI4SE benchmarks measure and the expectations and experience of the *practitioner* community [6]. We argue that this gap is not just a matter of imperfect correlation, but rather reflects *fundamental methodological flaws* in how many benchmarks are designed, evaluated, and maintained. This creates a misleading narrative of progress. In short, many popular benchmarks have only a tenuous connection to real-world software engineering.

The software engineering research community has repeatedly learned the consequences of flawed incentives. As two examples: fault localization research spent years optimizing for rank-based metrics that ultimately did not correspond to practical utility in real debugging tools; and, static analysis tools of the same era competed on number of reported defects, until the volume of low-value warnings led developers to disable them outright. Benchmarks should not incentivize the community into optimizing for the wrong targets.

This article distills our collective experience into a set of principled guidelines—dos and don'ts—for building, evaluating, and maintaining AI4SE benchmarks. We present these guidelines across five dimensions: defining the task, data gathering, data validation, measuring success, and long-term maintenance. Our goal is to provide a general framework for benchmark design in AI4SE, where benchmarks play a central role in reliably measuring and, in turn, defining scientific progress.

2 Defining the Task: External Validity

Before we start, it is imperative to identify what we are evaluating and why. This determines the population of similar tasks that the benchmark results are meant to more generally apply to.

Know your audience: A benchmark designer should begin by asking who the benchmark is for and what decisions it is meant to inform. A benchmark designed to measure general model code-reasoning abilities is very different from one designed to evaluate progress on a specific engineering subtask like debugging. Different goals imply a different audience, task design, evaluation protocol, and interpretation of results.

Clarify scope: Benchmarks should make explicit what kind of software engineering activity they represent, and what they do—and do not—attempt to capture. Without a precise task definition, benchmark results are easily interpreted more generally than they should. For instance, SWE-Bench is primarily an agentic program repair benchmark where passing tests indicate a successful patch. However, SWE-Bench improvements are sometimes taken as evidence of broader software engineering competence. Defining these operational boundaries precisely prevents scope creep. Results may apply more broadly only if the scope is also explicitly set more broadly, as in VIBench [10].

Ensure realistic scale and context: Even if the scope is clear, many widely used benchmarks capture only a narrow segment within that scope. For instance, a programming task benchmark like HumanEval [3] focuses on simple algorithmic problems implemented by a single function. They support controlled evaluation but do not reflect the scale, context, ambiguity, and interdependence of real-world software development. To this end, benchmarks like LongCodeArena [2] represent project-scale, workflow-oriented tasks.

3 Quality of Benchmark Tasks: Internal Validity

We must ensure that the benchmark results cannot be explained by factors that we did not explicitly control for. For instance, OpenAI gradually recommended and then decommissioned three versions

[1] https://cognition.com/blog/introducing-devin
[2] https://www.nist.gov/system/files/documents/2025/09/30/CAISI_Evaluation_of_DeepSeek_AI_Models.pdf

of SWE-Bench whenever new issues were identified. Benchmark quality has a huge impact on the interpretation of results, unbeknownst to practitioners and decision-makers who consume them.

Ensure data integrity: A persistent concern is data contamination or leakage. Tasks mined from open source repositories cannot realistically be deduplicated against a pre-training corpus in the foundation model era. One may attempt to measure leakage by proxy, e.g., by comparing n-gram accuracy on benchmark tasks to fresh tasks [9]. Even if we choose benchmark tasks beyond the training data cutoff date, temporal relationships may also bear on leakage potential. A repair benchmark released today may contain defects introduced and repaired in 2018, such that task and resolution are part of pre-training. To further mitigate data leakage, we should draw training, validation, and test splits over real-world repositories by project, not by file, function, or task instance. A good benchmark should therefore implement explicit contamination checks, document data provenance, and ideally provide a pipeline to support periodic addition of new data.

Match real-world distributions: Many benchmarks are constructed via a "convenience sampling" approach, i.e., scraping data from easily accessible sources, such as popular GitHub repositories, and focusing on easy-to-set-up task instances, e.g., from Python projects that can be automatically built and tested. Such benchmarks skew toward certain types of problems and programming languages, but fail to represent the diversity of software engineering work in the wild. Famously, SWE-Bench draws its samples from a small set of Python projects on GitHub.

Many benchmarks also ignore class imbalance, e.g., by providing as many vulnerable as non-vulnerable code snippets, while vulnerabilities are relatively rare in real codebases. Instead, benchmarks should strive to match the distributions of tasks, languages, and contexts that practitioners actually face, even if that requires more difficult-to-collect or less glamorous data. Duplication is itself a fact about real code, challenging this goal; selecting representatives from near-duplicate clusters, rather than discarding them, can preserve distribution with improved integrity.

Provide sufficient context: Task instances should provide access to the context a practitioner would naturally have in the work being simulated, such as the surrounding code base, project history, or failing tests. To minimize the influence from potentially confounding variables and maximize internal validity, the users of the benchmark must explicitly define which context is used and control for it. All other context should be ignored. Care must be taken to construct evaluation environments that exclude artifacts that would *not* be available in a realistic setting. For example, tasks constructed by checking out repositories at particular hashes while leaving the rest of the git history available, would leave the gold answer available to an enterprising agent. Benchmarks should moreover be transparent about what context is, or is not, included.

Manually validate the benchmark tasks: Automated pipelines can be efficient for gathering large volumes of data, but they are prone to errors, e.g., mislabeling, inclusion of irrelevant instances, or failure to capture real-world nuances. Manual validation, even if only on a sample of the data, is essential to catch these issues and to ensure that the benchmark reflects the intended task. Especially in software, many tasks are simply underdetermined, admitting several defensible resolutions with no single correct answer available for recovery. Such tasks cannot be served by more nuanced success criteria, they must simply be dropped. SWE-Bench provides a useful lesson: Its initial release of 2,294 instances contained many underspecified tasks, which were later refined into 500 manually validated instances in SWE-Bench Verified.

4 Quality of Task Grading: Internal and Construct Validity

The *grader for a task* decides whether the task was successful, or measures the degree to which the task was successful. Examples are ground truth labels in a classification task, test outcomes in automatic program repair, or rubrics in code reasoning. Even when the task is well specified, we need to examine its success criteria and the grader implementing them carefully.

Ensure that the success criteria are correct: We often have a high level understanding of what makes a task successful. Yet, for automatic grading we must determine operational success criteria. In program repair, we might consider a passing test suite as a successful repair task. However, because of the lack of complete coverage, flakiness, or environment dependencies, they are an imperfect proxy for full correctness. A repair may be plausible (and pass the entire test suite), but still incorrect. For other software engineering tasks, one may decide to use an LLM-as-a-judge. However, they introduce systemic biases, prompt sensitivity, and inconsistent behaviors. To increase validity, one should check alignment with expert human judgment when available, at least on a sample. During interpretation of the results, one should be aware of the limitations of the chosen success criteria. Benchmarks should overall report what the grader actually establishes. Passing the provided tests on 80% of instances is a different claim than resolving 80% of issues, and a leaderboard that reports the second while measuring the first overstates what its numbers support.

Ensure that the grader is correct: Even if we consider the success criteria as correct, we should ensure that our implementation to measure them is correct. For instance, when tests are involved, like in program repair, there should be sufficient test cases to indicate whether the bug is patched and no other bugs are introduced. When rubrics are inevitable, e.g., for code reasoning benchmarks [8], it is particularly important to minimize opportunities for "reward hacking".

Use the wisdom of the crowd: For certain tasks, e.g., code reviewing or code summarization, the ground truth is subjective or context-dependent. Leveraging the wisdom of the crowd through structured annotation can capture more nuanced and representative labels than a single expert or automated heuristic. Crowd-sourced validation can also refine a benchmark over time, as new insights emerge about what constitutes a valid instance or label. Crowdsourcing is also invaluable for training autoraters in scenarios where no deterministic verifier exists, such as greenfield tasks where no established ground truth or reference implementation is available to check against.

5 Measuring Success: From Leaderboard Scores to Meaningful Evidence

Perhaps the most consequential dimension of benchmark methodology is how success is measured. A single, publicly ranked number exerts enormous gravitational pull, regardless of whether that number is a valid proxy for the underlying capability. Closing that gap requires deliberate choices about metrics, statistical methods, and the claims the resulting numbers can legitimately support.

Consider multiple dimensions: Single-score leaderboards are appealingly simple, but they discard precisely the information that practitioners need to make informed decisions. For example, some users may care more about the worst-case performance of a model, rather than its average performance; others may care about the diversity of solutions it generates, the dollars spent per task solution, or its ability to explain its reasoning. Evaluating software qualities beyond functional correctness, such as maintainability or design quality, remains largely open. Benchmarks that aim to assess qualities besides behavior preservation should augment test passing with other measures (the software quality literature offers a number of imperfect proxies). To cater to a more diverse set of needs, benchmarks should offer multi-faceted evaluation with accompanying narratives rather than simply posting a number.

Evaluate naive baselines: A strong benchmark should include simple, interpretable baselines that any serious approach should beat. Which baseline is appropriate follows from the benchmark's goal, e.g., a random classifier for a detection task, a single-shot prompt for an agentic one, or retrieval alone for a RAG-based benchmark. Benchmarks can mislead community efforts without such baselines. For example, it was shown that a simple token-length based classifier can achieve surprisingly high accuracy on popular vulnerability detection benchmarks [7], suggesting that the benchmarks are not assessing genuine understanding. Baselines therefore provide both a bar for new techniques, and an instrument for assessing new benchmarks: a baseline that deliberately

withholds the signal the benchmark is meant to test should score poorly; if it instead scores well, this suggests that the benchmark's scoring mechanism is not yet reliable.

Provide variants of task instances: Publishing a single static version of a benchmark invites overfitting, memorization, and biased heuristics [9]. Instead, benchmarks should ensure robustness to variants of task instances, e.g., by providing meaning-preserving transformations of tasks. The mutations needed to test robustness must scale with model comprehension: simple perturbations that once revealed shortcut learning (e.g., adding an irrelevant comment to flip a label) no longer fool frontier models. Triggering similar failures now usually requires deeper changes—reworking control flow, renaming across scopes, or injecting adversarial defects—to test whether the model truly understands code or relies on surface patterns.

Ensure statistical rigor: LLM nondeterminism, coupled with metric variability (e.g., performance measurements) means that benchmark results are estimates. For example, a statistical analysis of SWE-Bench Verified results shows that the difference between two independent runs of the same approach easily exceeds the margin of improvement claimed by the top-ranked model over its predecessor [1]. A meaningful measure of success should therefore include confidence intervals, effect sizes, and appropriate statistical tests.

6 Long-term Maintenance: Benchmarks Are Not Artifacts, They Are Commitments

Current incentive structures treat benchmark creation as a one-time event. The paper gets cited, the leaderboard fills up, and the creators move on. What remains is a stale artifact that the community continues to trust long after the conditions that made it valid have changed.

Make the benchmark a living artifact: The central risk is "community overfitting"—tuning the entire research community to a fixed benchmark. For example, techniques over-optimized for Defects4J often reflect familiarity with its specific bugs rather than true fault localization capability. A solution would be actively refreshing the benchmark population on a declared schedule [4]. However, refresh challenges comparability; researchers frequently evaluate on older Defects4J releases to permit direct comparison against prior results. The SAT and MaxSAT competitions[3] offer an effective alternative: community-submitted problems accumulate under a stable API, and norms demand evaluating on recent problems. The benchmark thus becomes a standing community process, not a single static artifact.

Include a sunset clause: The ideal benchmark makes its own mortality explicit. Pre-declared obsolescence criteria, triggered by saturation of scores, shifts in the underlying software ecosystem, or evidence of systematic overfitting, transform maintenance from an afterthought into a design principle. We believe a sunset clause is not an admission of failure—it is a mark of methodological maturity, signaling that the benchmark was built to serve the field rather than to perpetuate itself.

7 Call to Action

AI4SE is too important to be guided by weak signals. Benchmarks must be treated as shared scientific infrastructure—rigorous, transparent, regularly refreshed, and aligned with the needs of both model trainers and engineering decision-makers.[4] Fundamentally, benchmarks define what a community counts as progress; measuring the wrong thing points the entire field at the wrong problems. Researchers, funders, and industry leaders should act now to make high-quality benchmarking a norm rather than an optional virtue. Doing so will move AI tooling beyond inflated claims toward evidence that genuinely drives better software and developer outcomes. The future of AI for software engineering is bright—but only if we measure it well.

[3] https://satcompetition.github.io/, https://maxsat-evaluations.github.io/
[4] https://cacm.acm.org/blogcacm/benchmarks-for-ai-in-software-engineering/

References

[1] Bjarni Haukur Bjarnason, Andr'e Silva, and Martin Monperrus. 2026. On randomness in agentic evals. In *Agents in the Wild Workshop, co-located with the International Conference on Learning Representations (ICLR)*.

[2] Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie Van Deursen, Maliheh Izadi, et al. 2024. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612* (2024).

[3] 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 preprint arXiv:2107.03374* (2021).

[4] Naman Jain, 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)*, Vol. 2025. 58791–58831.

[5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The 12th International Conference on Learning Representations (ICLR)*.

[6] Roham Koohestani, Philippe de Bekker, Begüm Koç, and Maliheh Izadi. 2025. Benchmarking AI Models in Software Engineering: A Review, Search Tool, and Unified Approach for Elevating Benchmark Quality. *IEEE Transactions on Software Engineering (IEEE TSE)* (2025).

[7] Jie Lin and David Mohaisen. 2024. Evaluating large language models in vulnerability detection under variable context windows. In *2024 International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 1131–1134.

[8] Ferida Mohammed, Fatma Ayad, Petros Maniatis, Satish Chandra, and Elizabeth Dinella. 2026. RubberDuckBench: A Benchmark for AI Coding Assistants. In *Proceedings of the 3rd International Workshop on Large Language Models for Code (LLM4Code '26)*. ACM, 113–120. doi:10.1145/3786181.3788710

[9] Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2025. Are large language models memorizing bug benchmarks?. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 1–8.

[10] Peter Zhong, Pashootan Vaezipoor, Fuyang Cui, Vaibhav Kumar, Azin Asgarian, James Austin, Toby Ho, Paul Inder, Imen Kedir, Zhen Li, Nick Ondo, Asna Shafiq, Ibrahim Sheikh, Edouard Sioufi, Setareh Soltanieh, Ben Wilde, Jacky Zhao, Ryan Carelli, Heather Miller, and Michele Catasta. 2026. ViBench: A Benchmark on Vibe Coding. In *ACM Conference on AI and Agentic Systems (ACM CAIS '26)*. ACM. doi:10.1145/3786335.3813162