Evaluating AI-Generated Code: A Taxonomy and Comparative Catalog of Vibe-Coding Benchmarks

Marko Horvat, Sven Leo Kupres
University of Zagreb
Faculty of Electrical Engineering and Computing
Department of Applied Computing
Unska 3, HR-10000 Zagreb, Croatia
{Marko.Horvat3, Sven-Leo.Kupres}@fer.unizg.hr

Abstract. *As Large Language Models and AI coding agents are increasingly integrated into software development processes through the novel vibe-coding paradigm, selecting appropriate benchmarks is becoming essential for assessing their correctness, security, efficiency, and practical utility. This paper systematizes the large and expanding landscape of benchmarks suitable for vibe-coding by providing a structured inventory of benchmark families used to evaluate AI-generated code. It introduces a compact six-dimensional taxonomy that classifies benchmarks according to intended use, evaluation scope, programming context, task type, assessment method, and practical applicability. The paper also presents methodological recommendations for selecting the most appropriate benchmark according to the employed software architectures and programming technologies, emphasizing that reliable assessment of modern AI coding systems requires an aggregated score from multiple complementary benchmarks rather than a single evaluation framework.*

Keywords. Artificial intelligence; Large Language Models; Software engineering; Computer programming; Benchmark testing; AI-assisted programming; Vibe-coding

1 Introduction

Generative AI and Large Language Models (LLMs) are rapidly disrupting software engineering by enabling code generation, repository editing, automated repair, test generation, and interactive debugging through natural language conversational and agentic interfaces. In practitioner discourse, in a conversational language of software developers, this workflow is increasingly described as *vibe-coding*: a style of software development in which the human developer specifies intent at a relatively high level, and the model autonomously generates computer code, proposes architectural solutions, and executes code artifacts or tools supporting software development (Horvat, 2025; Meske et al., 2025; Sarkar & Drosos, 2025). Although the term is informal rather than academic, the underlying evaluation problem is scientifically important: *how should one measure the quality of the AI-generated code?* Is it correct, secure, maintainable, context-aware, and useful for real-world software engineering work?

The rapid adoption of new AI technologies by the developer community raises serious concerns about reliability, architectural soundness, code explainability, long-term maintainability and, ultimately, the security of generated computer code (Elgendy et al., 2026). The models' ability to comprehend specific technical systems must be tested and objectively evaluated before they can be used reliably. While it has been demonstrated that LLMs can generate functional boilerplate code (Sapkota et al., 2025), particularly in domains such as web front-end (Horvat et al., 2025), and even backend development (Horvat et al., 2026), their ability to consistently make optimal decisions in software development remains the subject of active academic and professional research.

Early evaluations of code models relied primarily on compact programming problems, where a model receives a docstring or short natural language description and generates a function that passes hidden tests (Austin et al., 2021; M. Chen et al., 2021). These benchmarks remain useful because they are accessible, deterministic, and simple to replicate. However, their scope is limited. They rarely assess project context, multi-file dependencies, build systems, database access, front-end rendering, security, deployment assumptions, or long-term tool utilization. Later benchmark families expanded the evaluation target from function-level synthesis to pragmatic and repository-level software work, including LiveCodeBench, BigCodeBench, DevEval, SWE-bench, frontend benchmarks, backend benchmarks, secure code-generation benchmarks, and human-calibrated agentic tasks (Jain et al., 2025; Jimenez et al., 2024; J. Li et al., 2024; Rein et al., 2025; Shen et al., 2026; Vero et al., 2025; H. Zhu et al., 2025; Zhuo et al., 2025).

This paper examines the evaluation of AI-generated code using a taxonomy and comparative catalog of vibe-coding benchmarks. The resulting catalog of vibe-coding benchmarks includes only benchmark families

for which a traceable public source can be found, such as scientific publication, official repository, benchmark website, or technical documentation. This criterion is important because the benchmark landscape is rapidly expanding and contains numerous emerging frameworks whose scope, public availability, versioning, and evaluation criteria are not necessarily consistent.

The paper makes three distinct contributions. First, it provides a systematic inventory and structured list of the main benchmark families currently used to evaluate AI-generated code. Second, it provides concise methodological recommendations for selecting benchmark portfolios in scientific and industrial evaluations. Third, it proposes a compact taxonomy for the structured classification of vibe-coding benchmarks along six orthogonal dimensions: intended use, evaluation scope, programming context, task type, assessment method, and practical applicability.

2 Vibe-coding and AI-assisted coding

Vibe-coding is a novel software development paradigm in which a developer specifies the desired software functionality through natural language prompts addressed to a LLM, rather than implementing the software artifact through precise line-by-line programming (Horvat, 2025; Meske et al., 2025; Sarkar & Drosos, 2025). In this process, the developer acts primarily as a high-level coordinator who guides the AI coding agent through an iterative dialogue of code generation, testing, correction, and refinement (Horvat, 2026). As a result, the key challenge is no longer limited to mastering programming language syntax, understanding software libraries, or low-level implementation details but is transformed to efficiently articulating intent, decomposing requirements, evaluating generated artifacts, and detecting functional, architectural, security, or maintainability defects in the produced code. Vibe coding should therefore be distinguished from conventional AI-assisted programming, where the developer remains the principal author of the code and uses AI tools only to augment the manual coding process. In vibe coding, by contrast, the generated code may partly function as a black-box artifact whose quality depends on the interaction between natural language specifications, model capabilities, evaluation feedback, and corrective prompting. This makes reliable benchmarking especially important, since the assessment of vibe-coded software must extend beyond simple functional correctness and include repository-level behavior, security, efficiency, maintainability, and practical applicability.

The main difference between AI-assisted programming, or AI-assisted coding, and vibe coding is in the developer's role and level of responsibility for the generated code. In AI-assisted programming, the developer remains the primary author of the software artifact while AI tools are used to accelerate or improve certain elements of the manual coding process, such as code completion, refactoring, debugging, or test generation.

The generated code still needs to be understood, reviewed, and integrated by the human developer. In contrast, in vibe coding, the developer only expresses high-level intent in natural language and guides the LLM or AI agent through an iterative dialogue, with the model generating the majority or all of the software implementation.

In summary, the vibe-coding paradigm allows the human developer to operate without comprehending or even reading the programming code, whereas in AI-assisted programming, the developer is entirely aware of the code and can manually address various issues. Both approaches, vibe-coding and AI-assisted coding, produce AI-generated code and can benefit from the catalog presented in the paper.

3 AI-generated Code Benchmarks

Benchmarks for assessment of AI-generated code differ from standard natural language benchmarks used for text comparison (for example, to detect AI-generated text) because their outputs are not merely texts assessed for plausibility. They are executable artifacts that may compile, fail, pass tests, expose vulnerabilities, render a user interface, corrupt data, or interact with a live environment. For example, instead of merely checking how closely AI's text matches a human solution, OpenAI's HumanEval and HumanEval+ verify functional correctness by automatically executing the generated code against hidden unit tests. Consequently, thorough coding evaluation normally requires test oracles (i.e., providers of information that describe correct output): unit tests, integration tests, build logs, browser assertions, database state checks, proof-of-concept exploit tests, or carefully calibrated human and machine judging. Reviews of code-generation evaluation consistently point out that lexical similarity metrics such as exact-match, Levenshtein distance, BLEU, or ROUGE are insufficient when functional equivalence and software quality matter (Garcia & Chakravarty, 2024; Lu et al., 2021).

Typically, AI-code generation benchmarks use pass@k metric (Brown et al., 2024). This estimates the probability that at least one of the k code attempts generated by an AI model passes all the unit tests. Some other common metrics used in this context are test pass rate and executability rate (Paul et al., 2024). Code-centric structural and semantic metrics are used as well (for example, CodeBLEU, CodeBERTScore and CodeScore). However, even if code passes a unit test, it might be poorly written or inefficiently. Therefore, performance constraint metrics are also used, such as runtime speed (latency), memory footprint and cyclomatic complexity (Paul et al., 2024).

For the purposes of this overview, a vibe-coding benchmark is defined as a benchmark whose primary target is production, repair, completion, translation, or execution-grounded assessment of code or code-centered software development workflows.

This includes single-function synthesis, competitive programming, data science coding, repository editing, front-end UI generation, backend/API implementation, SQL generation, program repair, secure code generation, terminal-based agent tasks, and human-calibrated software work. This definition excludes purely conceptual programming questions, general natural language reasoning tasks, and code-understanding tasks that do not require the model to produce an executable or directly software-relevant artifact.

The literature review also shows that AI-coding benchmarks are not uniformly available across the entire software development life cycle. Development-phase tasks are more strongly represented, whereas requirements engineering, architecture design, release engineering, maintenance economics, and long-term quality assurance remain less systematically evaluated (K. Wang et al., 2026). This imbalance explains why high scores on popular coding benchmarks, such as HumanEval and MBPP, should not be interpreted as evidence of end-to-end software engineering competence.

4 Methodology

The comparative catalog of vibe-coding benchmarks was constructed in five steps. First, a comprehensive literature search was conducted using Google Scholar search engine with the following two sets of keywords: (*AI-assisted programming*, *AI-assisted coding*, *vibe-coding*) and (*assessment, evaluation, benchmark, evaluation benchmarks*). Each keyword from the first set was combined with each keyword from the second set, resulting in 12 two-term search queries. These searches generated a total of 349,290 results. For each query, the first 5 pages of Google Scholar results were examined, with each page containing 10 literature items. Papers that were accessible and relevant to the topic were then manually downloaded for further analysis and selection. The initial search results included HumanEval, MBPP, SWE-bench, CoderEval, LiveCodeBench, APPS, TACO, Android and mobile-related benchmarks, frontend benchmarks, backend benchmarks, and SQL/database benchmarks. Tag cloud generated from the retrieved literature is shown in Fig. 1.

In the second step, the retrieved literature was analyzed to establish benchmark frameworks for various technology stacks, including Spring Boot, Node.js, TypeScript, and PostgreSQL. The third step involved mapping each candidate benchmark to a primary public artifact, such as an archival paper, preprint, official repository, benchmark website, or dataset release. Fourth, benchmark families were identified for closely related variants that have the same core task formulation but differ in language coverage, test type, modality,



Figure 1: Tag cloud of key concepts and terms related to vibe-coding benchmarks and the evaluation of AI-generated code from the retrieved literature.

or validation method. In the fifth and final step, each family was described using a common schema based on their intended use, task type, input modality and granularity, programming language or domain, evaluation oracle and metric, availability and miscellaneous constraints, if they exist.

5 Taxonomy of Vibe-Coding Benchmarks

We propose a taxonomy in which the AI-assisted coding and vibe-coding benchmark landscape can be organized along six orthogonal dimensions. The first dimension in this taxonomy or classification is *task formulation* and includes the following software attributes: generation, completion, translation, repair, refinement, security hardening, test generation, or agentic workflow execution. The second dimension is *input modality* and applies to: natural language prompts, docstrings, partial code, issue reports, repository context, screenshots, diagrams, flowcharts, tests-as-specification, database schemas, or terminal states. The third dimension is *granularity* and covers: function, method, class, file, repository, application, or full workflow. The fourth dimension is *software domain*: algorithmic programming, data science, frontend, backend, mobile, SQL/database, spreadsheets, high-performance computing, security, or research replication. The fifth dimension is *oracle*: unit tests, compilation, integration tests, end-to-end browser execution, database checks, exploit validation, visual similarity, or human/LLM judging. The final sixth dimension is *evaluation metric* and is represented by AI-software evaluation metrics such as pass@k, execution accuracy, exact match, CodeBLEU, secure-pass variants, visual similarity, task success rate, human-time calibration, or economic value.

This classification was used as the methodological

structure for assembling the catalog of vibe-coding benchmarks provided below. Each benchmark identified in the literature review was analyzed and characterized according to the proposed dimensions, providing a common structure for describing otherwise heterogeneous benchmark families. The proposed taxonomy, which classifies vibe-coding benchmarks according to six orthogonal dimensions, is shown in Fig. 2.

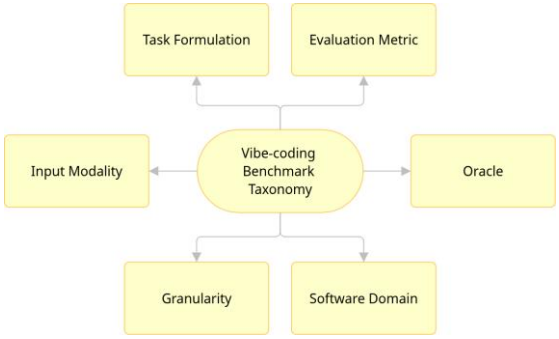


Figure 2: The proposed taxonomy of vibe-coding benchmarks defined by six orthogonal dimensions used in the construction of the benchmark catalog.

These dimensions reflect an important methodological trade-off. Function-level benchmarks may be simple to reproduce, scale, and execute repeatedly, but they do not completely reveal how software is developed, tested, and used in real-world settings. In contrast, repository-level, UI, mobile, backend, security, and terminal-based benchmarks more closely approximate realistic software engineering work, but they are also more expensive to execute, more sensitive to infrastructure differences, and more difficult to com-pare across evaluation environments. For this reason, comprehensive and reliable assessment of AI-generated code should rely on a portfolio of complementary benchmark families rather than a single benchmark score.

6 Comprehensive Benchmark Catalog

The complete catalog of AI-assisted programming and vibe-coding benchmarks is presented in three separate tables:

1. Table 1: Canonical code synthesis, competitive programming, multilingual, and data science benchmarks.
2. Table 2: Repository, UI, mobile, backend, database, and specialized-code benchmarks.
3. Table 3: Repair, agentic workflows, security, and human calibration benchmarks.

Table 1, Table 2, and Table 3 form one logical catalog split across three consecutive pages at the end of the paper to improve readability. Fig. 3 shows the timeline of initial release dates for selected representative benchmarks for AI-assisted coding listed in the catalog. The publication of benchmarks for assessing AI-generated program code has been steadily increasing following the introduction of LLMs, and this trend has been particularly evident since mid-2023. It should be noted that some benchmarks were developed long before the vibe coding paradigm has been introduced. Nevertheless, they remain relevant for evaluating code generated with this paradigm.

7 Recommendations for Benchmark Use

Based on the survey results, a comprehensive evaluation of AI-generated code should follow these five concise recommendations:

1. Use a set of benchmarks that combines at least one canonical synthesis benchmark, one contamination-aware or recent benchmark, and one domain-specific benchmark aligned with the intended software task.
2. When possible, use execution-grounded oracles. Additionally, report compilation, unit tests, integration tests, browser tests, database state checks, or exploit validation instead of relying on text similarity.
3. Report the harness (i.e., infrastructure that surrounds a software component to execute it) precisely. Repository-level and agentic benchmarks usually require specific versions of datasets, Docker images, library dependencies, different parameters, and tool access to be used successfully.
4. Separate correctness from quality. Functional success, maintainability, architectural soundness, readability, performance, and security are related but distinct dimensions.
5. Account for contamination and benchmark diffusion. Prefer fresh, time-split, private-test, or dynamically updated benchmarks when evaluating frontier models that may have seen public tasks during training or evaluation-time optimization.

For Spring Boot/PostgreSQL or Node.js/TypeScript/PostgreSQL backend evaluation, the implication is direct. Existing public benchmarks provide useful components: JavaBench for Java/OOP structure, Spider-style tasks for database query competence, BaxBench and ABC-Bench for backend testing, and SWE-bench-style tasks for repository repair. Additionally, a stack-specific evaluation should include a custom integration harness. Such a harness should compile generated code, start an ephemeral database, run framework-specific tests, validate serialization boundaries, and check software behavior relevant for security.

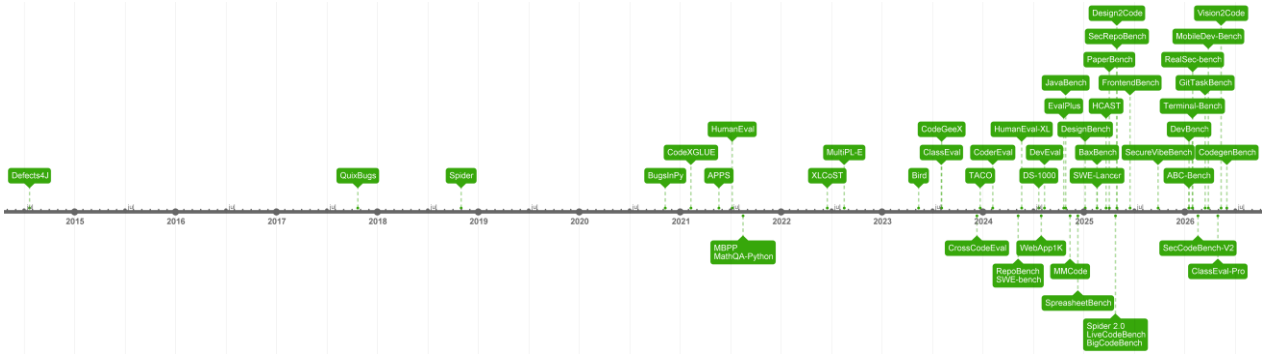


Figure 3: Timeline of first release dates for selected representative benchmarks for AI-assisted coding from the comparative catalog listed in Table 1, Table 2, and Table 3. The timeline clearly shows how the adoption of LLMs in code generation substantially accelerated the development of new benchmarks.

8 Discussion

The catalog shows a clear transition from narrow function synthesis to broader software engineering evaluation. Three observations are especially important for scientific benchmarking.

The most important observation is that *functional correctness is necessary but insufficient for a complete evaluation of AI-generated program code*. HumanEval, MBPP, EvalPlus, APPS, LiveCodeBench, BigCodeBench, and many repair benchmarks rely on unit tests as the primary oracle. This is methodologically stronger than string/text similarity because code may be lexically and syntactically different yet behaviorally equivalent. However, a test suite is only as strong as its coverage. EvalPlus was developed precisely because original HumanEval and MBPP tests could miss edge cases (J. Liu et al., 2023). In repository and back-end settings, test adequacy becomes even more difficult because models may pass local tests while violating different architectural, serialization, security, or performance requirements.

The second observation is that *context has become the most important factor*. Repository-level benchmarks show that coding ability depends on non-local information such as imports, APIs, naming conventions, build systems, framework idioms, and domain models. Consequently, CoderEval, RepoBench, CrossCodeEval, DevEval, JavaBench, and SWE-bench, for instance, are more congruent with professional development than just function synthesis (Cao et al., 2024; Ding et al., 2023; Jimenez et al., 2024; J. Li et al., 2024; T. Liu et al., 2024; H. Yu et al., 2024). This shift also introduces new methodological challenges. A model may fail due to a lack of reasoning ability, retrieval selecting incorrect files, dependencies not being installed, or the test harness differing from the original repository environment.

The third observation shows that *risk-sensitive evaluation is no longer an option*. Benchmarks such as BaxBench, SecRepoBench, RealSec-bench, SecureAgentBench, and SecCodeBench-V2 show that code can be correct under ordinary tests and still be insecure (J. Chen et al., 2025; L. Chen et al., 2026; Shen et al., 2026; Vero et al., 2025; Y. Wang et al., 2026). For backend, API, database, and mobile applications, a benchmark that ignores security can overestimate practical deployability. Software attributes such as security, privacy, authorization, injection resistance, dependencies, and usability should all be treated as equally important in applied environments as features evaluated by functional benchmarks.

9 Conclusion

Benchmarks for evaluating AI-generated code have evolved considerably beyond the first generation of short Python functions and have become highly complex. Given the speed with which large volumes of program code can be automatically generated using AI, benchmarks have become a necessity in the evaluation of vibe-coded software. Choosing the optimal benchmark has become very important but, at the same time, a difficult task.

HumanEval and MBPP benchmark families still remain valuable baselines, but modern vibe-coding evaluation must also address many additional modalities, such as repository context, multi-file dependencies, UI rendering, backend and database behavior, security, mobile ecosystems, terminal workflows, and human-calibrated software work. Therefore, *the correct methodological approach in the evaluation of AI-generated software is not a single benchmark score but an aggregated score based on a set of selected benchmarks aligned with the software features being tested*. To improve dissemination transparency and quality, such a joint score can be represented as a vector, with each individual score displayed separately, or, in a simpler way, as a scalar (i.e., a single real number) derived as a weighted sum of individual scores.

The most robust evaluation strategy should combine selected benchmarks with functional testing, testing strategies resistant to contamination, repository scaling tasks, and domain test oracles. For applied software engineering, additional dimensions such as security,

maintainability, performance, and integration correctness should be measured. The field of AI-generated code evaluation is converging towards more realistic and risk-sensitive assessment, but reproducibility, infrastructure cost, programming language imbalance, and benchmark contamination remain open challenges.

Acknowledgments

This paper is partly based on the research developed through the project "Maritime mAInd – Advanced platform for comprehensive design of marine vessels supported by machine learning algorithms" supported by the European Union within the Operational Programme Competitiveness and Cohesion in the financial period 2021–2027 (PK.1.1.12.0334).

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., & Mirhoseini, A. (2024). Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Cao, J., Chen, Z., Wu, J., Cheung, S.-C., & Xu, C. (2024). Javabench: A benchmark of object-oriented code generation for evaluating large language models. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 870–882.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., et al. (2022). Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Chen, J., Huang, H., Lyu, Y., An, J., Shi, J., Yang, C., Zhang, T., Tian, H., Li, Y., Li, Z., et al. (2025). Secureagentbench: Benchmarking secure code generation under realistic vulnerability scenarios. *arXiv preprint arXiv:2509.22097*.

Chen, L., Zhao, J., Cui, L., Su, T., Pan, X., Li, Z., Wu, Y., Cao, Q., Cai, Q., Zhang, J., et al. (2026). *Seccodebench-v2 technical report* (tech. rep.). Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda,

Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating large language models trained on code. https://arxiv.org/abs/2107. 03374

Chen, Y., Xie, C., Shi, Y., Zeng, W., Wang, Y., Zhang, H., & Gu, X. (2026). Classeval-pro: A cross-domain benchmark for class-level code generation. https://arxiv.org/abs/2604.26923

Cui, Y. (2024). Webapp1k: A practical code-generation benchmark for web app development. https://arxiv.org/abs/2408.00019

Ding, Y., Wang, Z., Ahmad, W., Ding, H., Tan, M., Jain, N., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., et al. (2023). Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, *36*, 46701–46723.

Elgendy, I. A., Dwivedi, Y. K., Al-Sharafi, M. A., Hosny, M., Helal, M. Y. I., Crick, T., Hughes, L., Alwahaishi, S., Mahmud, M., Dutot, V., & Al-Busaidi, A. S. (2026). Responsible vibe coding: Architecture, opportunities, and re-search agenda. *Journal of Computer Information Systems*, *0*(0), 1–19. https://doi.org/10. 1080/08874417.2026.2621186

Fakorede, M. A., Upadhyay, K., Siddique, A. B., & Farooq, U. (2026). Mobiledev-bench: A comprehensive benchmark for evaluating language models on mobile application development. https://arxiv.org/abs/2603.24946v1

FudanSELab. (2023). ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. https://github.com/ FudanSELab/ClassEval

Garcia, M., & Chakravarty, M. M. T. (2024). A re-view of evaluation methods for code generation. https://arxiv.org/abs/2408.16498

Gottumukkala, A. (2025). Kotlin-bench. https://github. com/firebenders/Kotlin-bench

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. (2021). Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Horvat, M. (2025). What is vibe coding and when should you use it (or not)? *Authorea Preprints.*

Horvat, M. (2026). Which prompting technique is better? Improving vibe coding code quality with efficient prompt design for web front-end development. *interaction*, *9*, 10.

Horvat, M., Kralj, B., & Gledec, G. (2025). A comparative study of vibe coding with chatgpt and gemini in front-end web development. *Proceedings of the 36th International Scientific Conference: Central European Conference on Information and Intelligent Systems: CECIIS 2025*, 787–796.

Horvat, M., Ursić, I., & Krmpotić, K. (2026). Comparative evaluation of gemini and deepseek for llm-generated code quality and architectural robustness in backend software engineering. *Electronics*, *15*(13). https://doi.org/10.3390/ electronics15132805

Jain, N., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., & Stoica, I. (2025). Livecodebench: Holistic and contamination free evaluation of large language models for code. *International Conference on Learning Representations*, *2025*, 58791–58831.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press,

O., & Narasimhan, K. (2024). Swe-bench: Can language models resolve real-world github issues? *International Conference on Learning Representations, 2024*, 54107–54157.

Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4j: A database of existing faults to enable controlled testing studies for java programs. *Proceedings of the 2014 international symposium on software testing and analysis*, 437–440.

Kumarappan, A., Golnari, P. A., Wen, W., Liu, X., Ryan, G., Sun, Y., Fu, S., & Nallipogu, E. (2026). De-vbench: A realistic, developer-informed benchmark for code generation models. https://arxiv. org/abs/2601.11895

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, S. W.-t., Fried, D., Wang, S., & Yu, T. (2022). Ds-1000: A natural and reliable benchmark for data science code generation. https://arxiv.org/abs/2211.11501

Lei, F., Chen, J., Ye, Y., Cao, R., Shin, D., Su, H., Suo, Z., Gao, H., Hu, W., Yin, P., et al. (2025). Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *International Conference on Learning Representations, 2025*, 28691–28735.

Li, J., Li, G., Zhao, Y., Li, Y., Liu, H., Zhu, H., Wang, L., Liu, K., Fang, Z., Wang, L., et al. (2024). Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *Findings of the Association for Computational Linguistics: ACL 2024*, 3603–3614.

Li, J., Wu, W., Hu, J., Zheng, Q., Wu, B., Zheng, J., Lu, Y., & Fu, H. (2026). Codegenbench: Can llms write efficient code across architectures? https://arxiv.org/abs/2606.04023

Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Cao, R., Geng, R., Huo, N., Zhou, X., Ma, C., Li, G., Chang, K. C. C., Huang, F., Cheng, R., & Li, Y. (2023). Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. https://arxiv.org/abs/2305.03111

Li, K., Tian, Y., Hu, Q., Luo, Z., Huang, Z., & Ma, J. (2024). Mmcode: Benchmarking multimodal large language models for code generation with visually rich programming problems. *Findings of the Association for Computational Linguistics: EMNLP 2024*, 736–783.

Li, R., Fu, J., Zhang, B.-W., Huang, T., Sun, Z., Lyu, C., Liu, G., Jin, Z., & Li, G. (2023). Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017). Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 55–56.

Liu, J., et al. (2023). EvalPlus: A rigorous evaluation framework for LLM4Code. https://github.com/evalplus/evalplus

Liu, T., Xu, C., & McAuley, J. (2024). Repobench: Benchmarking repository-level code auto-completion systems. *International Conference on Learning Representations, 2024*, 47832–47850.

Lopez-Latorre, Z. (2026). Android bench. https://github.com/android-bench/android-bench

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Ma, Z., Zhang, B., Zhang, J., Yu, J., Zhang, X., Zhang, X., Luo, S., Wang, X., & Tang, J. (2024). Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *Advances in Neural Information Processing Systems*, *37*, 94871–94908.

Maninger, D., Chemnitz, L., Sharifloo, A. M., Brugger, J., & Mezini, M. (2025). Benchmarking web api integration code generation. *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, 240–248.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., Shen, J., Ye, G., Lin, H., Poulos, J., Wang, M., Nezhurina, M., Jitsev, J., Lu, D., Mastromichalakis, O. M., . . . Schmidt, L. (2026). Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. https://arxiv.org/abs/2601.11868

Meske, C., Hermanns, T., Von der Weiden, E., Loser, K.U., & Berger, T. (2025). Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. *IEEE Access*, *13*, 213242–213259.

Miserendino, S., Wang, M., Patwardhan, T., & Hei-decke, J. (2025). Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering? https://arxiv.org/abs/2502.12115

Nam, J., et al. (2025). Autobe - ai backend builder for prototype to production. https://github.com/wrtnlabs/autobe

Ni, Z., Wang, H., Zhang, S., Lu, S., He, Z., Tang, Z., Hu, S., Li, B., Hu, C., Jiao, B., et al. (2026). Git-taskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. *Proceedings of the AAAI Conference on Artificial Intelligence*, *40*(38), 32564–32572.

Paul, D. G., Zhu, H., & Bayley, I. (2024). Benchmarks and metrics for evaluations of code generation: A critical review. *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, 87–94.

Peng, Q., Chai, Y., & Li, X. (2024). Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *Proceedings of the 2024 joint international conference on computational linguistics, language resources and evaluation (LREC-COLING 2024)*, 8383–8394.

Periasami, A. V., Wang, J., & Dhingra, B. (2026). Vision2code: A multi-domain benchmark for

evaluating image-to-code generation. https://arxiv.org/abs/2605.11307

Princeton NLP. (2024). SWE-bench official repository. https://github.com/princeton-nlp/SWE-bench

Rein, D., Becker, J., Deng, A., Nix, S., Canal, C., O'Connel, D., Arnott, P., Bloom, R., Broadley, T., Garcia, K., et al. (2025). Hcast: Human-calibrated autonomy software tasks. *arXiv preprint arXiv:2503.17354*.

Sapkota, R., Roumeliotis, K. I., & Karkee, M. (2025). Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*.

Sarkar, A., & Drosos, I. (2025). Vibe coding: Programming through conversation with artificial intelligence. *arXiv preprint arXiv:2506.23253*.

Shen, C., Dilgren, C., Chiniya, P., Griffith, L., Ding, Y., & Chen, Y. (2026). Secrepobench: Benchmarking code agents for secure code completion in real-world repositories. https://arxiv.org/abs/ 2504.21205

Si, C., Zhang, Y., Li, R., Yang, Z., Liu, R., & Yang, D. (2025). Design2code: Benchmarking multimodal code generation for automated front-end engineering. *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 3956–3974.

Starace, G., Jaffe, O., Sherburn, D., Aung, J., Chan, J. S., Maksin, L., Dias, R., Mays, E., Kinsella, B., Thompson, W., et al. (2025). Paperbench: Evaluating ai's ability to replicate ai research. *arXiv preprint arXiv:2504.01848*.

Vero, M., Mündler, N., Chibotaru, V., Raychev, V., Baader, M., Jovanović, N., He, J., & Vechev, M. (2025). Baxbench: Can llms generate correct and secure backends? *arXiv preprint arXiv:2502.11844*.

Wang, K., Li, T., Zhang, X., Wang, C., Sun, W., Liu, Y., Liu, A., Liu, X., Shen, C., & Shi, B. (2026). Software development life cycle perspective: A survey of benchmarks for code large language models and agents. https://arxiv.org/abs/2505.05283

Wang, Y., Zhang, Z., Wang, C., Xu, X., Liu, M., Wang, Y., Chen, J., & Zheng, Z. (2026). Realsec-bench: A benchmark for evaluating secure code generation in real-world repositories. *arXiv preprint arXiv:2601.22706*.

Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J. E., Yieh, Y., et al. (2020). Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 1556–1560.

Xiao, J., Wang, M., Lam, M. H., Wan, Y., Liu, J., Huo, Y., & Lyu, M. R. (2025). Designbench: A comprehensive benchmark for mllm-based front-end code generation. *arXiv preprint arXiv:2506.06251*.

Yang, J., Guo, H., Ji, L., Zhou, J., Zheng, R., Lei, Z., Zhang, S., Xi, Z., Liu, S., Wang, Y., Wang, B., Zheng, Y., Gui, T., & Qiu, X. (2026). Abcbench: Benchmarking agentic backend coding in real-world development. https://arxiv.org/ abs/2601.11077

Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Liang, G., Li, Y., Wang, Q., & Xie, T. (2024). Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–12.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., et al. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Proceedings of the 2018 conference on empirical methods in natural language processing*, 3911–3921.

Zhang, F., Wu, L., Bai, H., Lin, G., Li, X., Yu, X., Wang, Y., Chen, B., & Keung, J. (2025). Humaneval-v: Benchmarking high-level visual reasoning with complex diagrams in coding tasks. https://arxiv.org/abs/2410.12381

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y., et al. (2023). Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x, 5673–5684.

Zhu, H., Zhang, Y., Zhao, B., Ding, J., Liu, S., Liu, T., Wang, D., Liu, Y., & Li, Z. (2025). Frontend-bench: A benchmark for evaluating llms on front-end development via automatic evaluation. *arXiv preprint arXiv:2506.13832*.

Zhu, M., Jain, A., Suresh, K., Ravindran, R., Tipirneni, S., & Reddy, C. K. (2022). Xlcost: A benchmark dataset for cross-lingual code intelligence. https://arxiv.org/abs/2206.08474

Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I., et al. (2025). Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *International Conference on Learning Representations*, *2025*, 66602–66656.

Table 1: Comprehensive catalog of AI-generated code benchmarks, part I: canonical code synthesis, competitive programming, multilingual, and data-science benchmarks.

Benchmark family	Intended use	Input and granularity	Domain/languages	Oracle and metrics	Key properties and limitations
HumanEval family: HumanEval, HumanEval+, HumanEval-X, HumanEval-XL, HumanEval-V	Canonical code synthesis; strengthened, multilingual, and visual variants	Docstring or natural-language prompt to function; HumanEval-V adds visual input	Python baseline; variants cover additional programming and natural languages	Hidden unit tests; pass@k; stronger tests in EvalPlus	Reproducible and historically central, but small, Python-centered in the original form, and vulnerable to contamination; variants improve test strength, language coverage, and visual reasoning (M. Chen et al., 2021; J. Liu et al., 2023; Peng et al., 2024; Zhang et al., 2025; Zheng et al., 2023).
MBPP family: MBPP, MBPP+	Entry-level standalone program synthesis	Short natural-language problem descriptions to small programs	Python, beginner-to-intermediate algorithmic tasks	Unit tests; pass@k; strengthened tests in MBPP+	Useful for quick baseline evaluation and novice-style tasks; less realistic than project-level benchmarks and limited in language/ecosystem coverage (Austin et al., 2021; J. Liu et al., 2023).
AixBench	Method-level code generation with automatic and manual assessment	Natural-language or specification-like prompt to method implementation	General programming tasks	Automatic correctness metric and manual evaluation	Bridges pure unit-test evaluation and qualitative assessment; smaller and less widely adopted than HumanEval or MBPP.
APPS, TACO, LiveCodeBench, CodeElo	Competitive-programming and algorithmic reasoning evaluation	Long problem statement to complete solution; some variants include repair or execution prediction	Algorithmic programming, mainly contest-style settings	Hidden tests, contest submission, pass@k, Elo-style scoring	More difficult and contamination-aware than static function suites, especially LiveCodeBench; still measures contest programming more than industrial software engineering (Hendrycks et al., 2021; Jain et al., 2025; R. Li et al., 2023).
BigCodeBench, BigCodeBench-Hard, CodeMixBench	Practical function generation requiring library and API use	Instruction or docstring to function using many libraries and complex constraints	Python-centered practical coding across library domains	Unit tests with stronger branch coverage; pass@k	Better reflects tool/library use than HumanEval; still mostly function-level and sensitive to execution environment availability (Zhuo et al., 2025).
DS-1000, DSBench	Data-science code generation and agentic data-analysis tasks	StackOverflow-style or data-analysis prompt plus data context	Python data science, scientific and analytics libraries	Unit tests, output comparison, analy-sis/modeling task completion	Strong domain realism for data science; narrower transferability to general software engineering (Lai et al., 2022).
CodeXGLUE, XLCoST	Broad code-intelligence suites, including generation, translation, summarization, search, and refinement	Natural-language-to-code, code-to-code, code-to-text, and related formats	Multiple languages and code-intelligence tasks	Mixed metrics: BLEU, CodeBLEU, exact match, retrieval and task-specific scores	Important for broad benchmarking but not all subtasks are execution-grounded; scores should not be equated with functional correctness (Lu et al., 2021; M. Zhu et al., 2022).
AVATAR, MultiPL-E, McEval, MCoNaLa	Code translation and multilingual natural-language-to-code evaluation	Parallel code, translated prompts, multilingual natural-language inputs	Java/Python and many additional programming/natural languages	Unit tests where available; translation and code-similarity metrics in some settings	Addresses English/Python monoculture; automatic translation can introduce artifacts and may not capture idiomatic language-specific programming (Cassano et al., 2022).

Table 2: Comprehensive catalog of AI-generated code benchmarks, part II: repository, UI, mobile, backend, database, and specialized-code benchmarks.

Benchmark family	Intended use	Input and granularity	Domain/languages	Oracle and metrics	Key properties and limitations
MMCode, Code-Vision, Vision2Code	Multimodal code generation from visual problem statements, flowcharts, diagrams, or other visual sources	Image plus optional text to executable code or rendered artifact	Multimodal programming and visual reasoning	Unit tests, visual similarity, VLM or human judging depending on task	Captures visual specification ability absent from text-only benchmarks; evaluation may depend on rendering and judge quality (K. Li et al., 2024; Periasami et al., 2026).
CoderEval, RepoBench, CrossCodeEval, DevEval, DevBench	Pragmatic generation and completion using project context, repository dependencies, or telemetry-derived developer tasks	Partial code, repository context, local APIs, or development traces; method/file/repo level	Python, Java, TypeScript, C#, and mixed repositories	Unit tests, exact match, retrieval utility, task success, and judge-assisted usefulness	Measures non-local context use and dependency awareness; results can be confounded by retrieval quality, environment setup, and repository availability (Ding et al., 2023; Kumarappan et al., 2026; J. Li et al., 2024; T. Liu et al., 2024; H. Yu et al., 2024).
ClassEval, ClassEval-Pro, JavaBench	Class-level and project-level structured code generation	Class skeletons, method descriptions, signatures, and project context	Python classes; Java OOP/project settings	Unit tests, class-wise and method-wise pass@k	Bridges single functions and full repositories; useful for OOP and enterprise-like structure, but less comprehensive than repository repair benchmarks (Cao et al., 2024; Y. Chen et al., 2026; FudanSELab, 2023).
Design2Code, DesignBench, FrontendBench, WebSight/WebSight-Test	Frontend, UI, screenshot-to-code, and interactive web-component evaluation	Screenshot, design image, prompt, or component specification to HTML/CSS/JS/framework code	Web UI; React, Vue, Angular, vanilla HTML/CSS in some variants	Rendering similarity, browser tests, end-to-end assertions, human/VLM agreement	Essential for UI coding; deterministic testing is harder because visual fidelity, layout similarity, and interactivity must be evaluated jointly (Si et al., 2025; Xiao et al., 2025; H. Zhu et al., 2025).
WebApp1K, BaxBench, ABC-Bench, Web API Integration benchmark	Web application, backend, and API-integration code generation	Application requirements, tests-as-specification, API descriptions, or backend tasks to multi-file code	Web applications, backend services, API clients	Unit/integration tests, API checks, exploit/security validation in BaxBench	Stronger ecological validity for production-style development; setup is framework-sensitive and more expensive than function-level evaluation (Cui, 2024; Maninger et al., 2025; Vero et al., 2025; Yang et al., 2026).
Spider, BIRD, Spider 2.0	Text-to-SQL and database-grounded code generation	Natural-language question plus schema to SQL or SQL-agent workflow	Relational databases and enterprise-like data access	Execution accuracy, exact match, SQL-specific metrics	Crucial for database code, but SQL-specific results should not be generalized uncritically to full backend software engineering (Lei et al., 2025; J. Li et al., 2023; T. Yu et al., 2018).
SpreadsheetBench, CodegenBench	Spreadsheet automation and performance-oriented code generation	Natural-language spreadsheet task or kernel-generation specification	Spreadsheets; BLAS/HPC and architecture-specific code	File-level tests, execution correctness, performance metrics	Expands evaluation beyond general-purpose scripts; highly domain-specific and infrastructure-dependent (J. Li et al., 2026; Ma et al., 2024).
Android Bench, Kotlin-Bench, AutoBe	Mobile and backend evaluation	Mobile/backend-oriented benchmarks	Android, Kotlin, backend development		(Gottumukkala, 2025; Lopez-Latorre, 2026; Nam et al., 2025)

Table 3: Comprehensive catalog of AI-generated code benchmarks, part III: repair, agentic workflows, security, and human calibration.

Benchmark family	Intended use	Input and granularity	Domain/languages	Oracle and metrics	Key properties and limitations
Defects4J, Defects4J-NL2Fix, BugsInPy, QuixBugs	Automated program repair and bug-fix generation	Buggy code, tests, and sometimes natural-language issue text to patch	Java, Python, and small algorithmic programs	Unit tests, patch correctness, top-*k* repair accuracy	Foundational for repair; test-suite overfitting and incomplete specifications remain classic threats (Just et al., 2014; Lin et al., 2017; Widyasari et al., 2020).
SWE-bench family: SWE-bench, Lite, Verified, Multimodal, Multi-SWE-bench, SWE-Bench Pro, MobileDev-Bench	Real-world issue resolution over repositories; increasingly multimodal, multilingual, long-horizon, and mobile variants	Issue report plus repository, build/test environment, sometimes screenshots or mobile app context	Python core benchmark; variants cover JavaScript, multiple languages, and mobile ecosystems	Build-and-test harnesses, issue-resolution rate, pass@1, patch success	Reference family for coding agents; high ecological validity but expensive, environment-sensitive, and increasingly exposed to benchmark diffusion (Fakorede et al., 2026; Jimenez et al., 2024; Princeton NLP, 2024).
GitTaskBench, Terminal-Bench, GitGoodBench	End-to-end repository and terminal workflows for code agents	Natural-language task plus repository or terminal environment	GitHub tasks, CLI tasks, software maintenance workflows	Task success, com-mand/environment checks, cost-aware or alpha-style metrics	Strong measure of tool-using autonomy; difficult to reproduce and compare because task execution depends on environment state (Merrill et al., 2026; Ni et al., 2026).
SecRepoBench, SecureAgentBench, RealSec-bench, SecCodeBench-V2	Correct-and-secure code generation, secure repair, and vulnerability-aware repository tasks	Security-focused prompts, vulnerable repositories, project scaffolds, or exploit scenarios	C/C++, Java, and multi-language industrial settings	Unit tests plus static analysis, exploit proof-of-concept, secure-pass metrics	Shows that functional correctness and security diverge; security must be evaluated explicitly rather than inferred from ordinary tests (J. Chen et al., 2025; L. Chen et al., 2026; Shen et al., 2026; Y. Wang et al., 2026).
SWE-Lancer, HCAST, PaperBench	Human-calibrated, economically grounded, or research-replication software work	Freelance software tasks, calibrated human baselines, or paper-replication subtasks	Software engineering, ML engineering, cybersecurity, scientific code	Task success, monetary value, human-time calibration, rubric grading	Closer to real value and effort than pass@k; costly, complex, and sometimes judge-dependent (Miserendino et al., 2025; Rein et al., 2025; Starace et al., 2025).