



International Journal of Multidisciplinary Conference Proceedings

✉ editor@ijmcp.com

🌐 <https://www.ijmcp.com>

A Comparative Evaluation of Large Language Models for Automated Code Generation and Debugging

1stMuhammad Zulqarnain Siddiqui, 2ndKamran Khan

1st Associate Professor, Department of Computer Science, Iqra University, Karachi, Pakistan

2nd Associate Professor, Department of Computer Science, Iqra University, Karachi, Pakistan

KEYWORDS	ABSTRACT
Large Language Models, Code Generation, Automated Debugging, Software Engineering, HumanEval, DebugBench, SWE-bench.	Large language models (LLMs) have moved from text completion systems to practical programming assistants capable of generating functions, explaining code, locating defects, and proposing repairs. Their apparent fluency, however, does not translate uniformly into functional correctness across programming settings. This paper presents a structured comparative evaluation of published benchmark evidence on automated code generation and debugging. The analysis integrates results from HumanEval, MBPP, QuixBugs, DebugBench, SWE-bench, LiveCodeBench, and competition-level code generation studies, with attention to test-based correctness, repair success, task complexity, runtime feedback, and benchmark contamination. The evidence shows a clear capability gradient. LLMs perform strongly on short, self-contained programming tasks, and repeated sampling substantially increases the chance of obtaining a correct solution. Debugging performance is also substantial for syntax and reference errors, while logic defects and multiple simultaneous bugs remain more difficult. Repository-level software engineering creates a sharper challenge because successful repair requires long-context understanding, multi-file reasoning, environment interaction, and reliable test execution. The comparison further shows that benchmark design strongly shapes reported performance, making direct score aggregation inappropriate across datasets. Overall, modern LLMs function effectively as code-generation and debugging assistants, but their outputs still require executable validation, security review, and human oversight before integration into production software.
ARTICLE HISTORY Date of Preparation: 30-10-2025	
Conference Organizer(s) Research Consultancy on Social & Management Development & University of Lahore, Sargodha Campus, Pakistan	
Corresponding Email	
Volume-Issue-Page Number	kamrankhaniu@gmail.com 3(1), 32
Citation	Siddiqui, M. Z., & Khan, K. (2025). <i>A comparative evaluation of large language models for automated code generation and debugging</i>. <i>International Journal of Multidisciplinary Conference Proceedings</i>, 3(1), 32

1. Introduction

Large language models have become a central technology in contemporary software engineering. Models trained on natural language and source code can translate requirements into executable programs, complete partially written functions, generate unit tests, explain unfamiliar code, and suggest repairs for defective programs. This shift extends earlier work in program synthesis by replacing narrowly specified search spaces with probabilistic models that learn programming patterns from large corpora. HumanEval and MBPP established early evidence that code-focused LLMs could solve short programming problems from natural-language descriptions, while CodeGen and AlphaCode showed that scale, code-specialized training, sampling, and behavioral filtering can extend performance to more demanding synthesis tasks (Austin et al., 2021; Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2023).

The practical attraction of LLM-based coding systems is straightforward: they reduce the distance between intent and implementation. A programmer can describe a function, provide a partial code context, or paste a failing program and receive a candidate solution within seconds. Empirical studies of GitHub Copilot also show that developers often value generated code as a useful starting point even when it does not improve every measure of task completion (Vaithilingam et al., 2022). At the same time, code that looks plausible can still be functionally wrong, insecure, inefficient, or inconsistent with a larger codebase. Pearce et al. (2022), for example, found substantial security weaknesses in generated code across high-risk cybersecurity scenarios, while Moradi Dakhel et al. (2023) reported that Copilot can produce efficient solutions but also buggy and non-reproducible outputs.

Code generation and debugging therefore require different but related forms of reasoning. Generation begins from a specification and constructs a program that satisfies tests or expected behavior. Debugging starts from an existing program that fails and requires the model to identify the defect, preserve correct behavior, and modify only what is necessary. The debugging task can be easier when the error is local and visible, such as a missing delimiter or invalid identifier, but it can be harder when the defect is logical, distributed across functions, or combined with other errors. DebugBench provides strong evidence for this distinction by evaluating 4,253 debugging instances across syntax, reference, logic, and multiple-error categories in C++, Java, and Python (Tian et al., 2024).

Another challenge concerns the meaning of a benchmark score. HumanEval measures function-level synthesis from docstrings, whereas SWE-bench requires a model to understand real GitHub issues and modify repository code so that the relevant test suite passes. LiveCodeBench further addresses contamination and saturation by continuously collecting newer programming problems and evaluating broader capabilities, including generation, self-repair, execution, and output prediction (Jain et al., 2025). Consequently, a high score on a short-form benchmark does not imply comparable performance on repository-level maintenance.

Against this background, this paper compares verified empirical evidence across major code-generation and debugging benchmarks. The analysis addresses three questions: (1) how strongly do LLMs perform on function-level code generation, (2) how does debugging performance vary by error type and feedback, and (3) what happens when evaluation moves from isolated functions to real software repositories? The aim is not to create a single league table from incompatible datasets. Instead, the paper interprets model performance in relation to the task structure, evaluation metric, and software-engineering context in which each result was produced.

2. Literature Review

2.1 LLM-Based Code Generation

Early LLM research established that programming ability emerges strongly when language models are exposed to source-code corpora. Chen et al. (2021) introduced Codex and HumanEval, a set of 164 hand-written Python programming problems evaluated through unit tests. The study emphasized functional correctness rather than textual similarity, an important distinction because several syntactically different programs can solve the same task. Codex achieved a 28.8% pass@1 score on HumanEval, and repeated sampling increased pass@100 to 70.2%. This result established two durable

insights: single-sample accuracy understates the search capability of generative models, and executable tests provide a more meaningful evaluation than surface-level similarity.

Austin et al. (2021) introduced MBPP, containing 974 entry-level Python programming tasks, and found that program-synthesis performance improved systematically with model scale. Nijkamp et al. (2023) extended open code generation through CodeGen and showed that decomposing a programming intention into multiple conversational turns can improve synthesis. AlphaCode pushed the problem toward competitive programming. In simulated Codeforces contests, it reached an average ranking within the top 54.3% of participants, demonstrating that large-scale sampling and filtering can support more complex algorithmic reasoning (Li et al., 2022). These studies collectively shifted code generation from completion toward testable program synthesis.

2.2 Automated Debugging and Program Repair

Automated program repair predates LLMs, but conversational models introduce a flexible repair interface because a user can provide failing code, error messages, expected outputs, or additional constraints. Sobania et al. (2023) evaluated ChatGPT on the QuixBugs benchmark and found that conversational feedback materially improved repair performance, reaching 31 correct repairs out of 40 bugs when additional hints were provided. This interaction pattern is important because real debugging rarely occurs from a single static prompt; developers inspect traces, rerun tests, and refine hypotheses.

Tian et al. (2024) addressed limitations of earlier small debugging studies through DebugBench. The benchmark contains 4,253 instances across 18 minor bug types and three languages. GPT-4-0613 passed 75.0% of the bug instances and GPT-3.5-turbo-0613 passed 62.1%, while the tested open models generally scored lower. The error taxonomy also revealed a consistent difficulty gradient: syntax and reference defects were easier than logic defects, and programs containing multiple simultaneous bugs were especially difficult. Runtime feedback improved syntax and reference repair but did not uniformly help logical debugging, indicating that low-level traces cannot substitute for semantic reasoning.

2.3 From Functions to Repositories

The strongest caution against overgeneralizing from small benchmarks comes from repository-level evaluation. SWE-bench contains 2,294 real software-engineering issues drawn from 12 Python repositories. Each task requires understanding an issue description, navigating an existing codebase, editing relevant files, and satisfying project tests. In the original evaluation, the best model, Claude 2, resolved only 1.96% of issues (Jimenez et al., 2024). The gap between this result and short-form coding benchmarks shows that realistic software maintenance introduces context management, dependency reasoning, tool use, and multi-file coordination as separate sources of failure.

LiveCodeBench complements this perspective by focusing on temporal freshness and contamination control. Jain et al. (2025) evaluated more than 50 models on newly collected programming problems and demonstrated that established coding benchmarks can become saturated or contaminated over time. The benchmark also broadens evaluation beyond generation to self-repair, code execution, and test-output prediction. Recent systematic reviews reach the same general conclusion: LLMs have broad value across software-engineering activities, but performance remains highly task dependent and requires rigorous evaluation, validation, and workflow design (Hou et al., 2024).

2.4 Evaluation Reliability, Security, and Human Factors

Benchmark accuracy is only one component of software quality. Generated programs can pass visible tests while violating security, maintainability, or dependency constraints. Pearce et al. (2022) examined Copilot outputs across scenarios linked to high-risk Common Weakness Enumeration categories and found that approximately 40% of the generated programs were vulnerable. This result is especially relevant to comparative evaluation because models trained on large public repositories can reproduce insecure programming patterns together with useful ones. A test-driven assessment of functional correctness should therefore be paired with security-oriented analysis when the generated code handles authentication, cryptography, input validation, memory, or external resources.

Human interaction also changes the effective performance of a coding model. Vaithilingam et al. (2022) observed that programmers valued Copilot as a source of starting points, yet participants also encountered difficulty understanding and debugging generated snippets. Moradi Dakhel et al. (2023) reached a related conclusion: generated solutions could be efficient and comparable with human contributions in some dimensions, but buggy and non-reproducible suggestions remained common enough to create risk, particularly for less experienced users. These findings indicate that coding LLMs should be evaluated as components of a human-software workflow rather than as isolated text generators.

A further reliability issue is benchmark contamination. Widely circulated datasets can appear in training corpora, allowing a model to reproduce previously seen solutions instead of demonstrating generalizable program reasoning. LiveCodeBench addresses this concern through newer, time-segmented problems and reports evidence of saturation and contamination in older benchmarks (Jain et al., 2025). The implication for comparative research is that benchmark freshness, release date, model training cutoff, and executable test coverage are part of the evaluation design rather than peripheral metadata.

3. Methodology

The study uses a structured comparative benchmark synthesis. Evidence was included when a study or benchmark met four criteria: it evaluated an LLM or LLM-based coding system; it addressed code generation, debugging, or repository-level repair; correctness was assessed through executable tests, accepted solutions, or an explicitly defined repair metric; and the source provided enough methodological detail to interpret the result. Peer-reviewed conference and journal papers formed the core evidence base, supplemented by foundational arXiv papers when they introduced widely adopted benchmarks such as HumanEval and MBPP.

The comparison centers on six benchmark families: HumanEval, MBPP, competitive-programming evaluation used for AlphaCode, QuixBugs, DebugBench, and SWE-bench, with LiveCodeBench used to interpret benchmark freshness and contamination. Extracted dimensions included task granularity, programming context, model identity, correctness metric, role of sampling or feedback, and the reported performance outcome. The synthesis preserves each benchmark's original metric. Scores from different benchmarks are not averaged because pass@k, percentage of repaired bugs, competition ranking, and repository issue-resolution rate measure different constructs.

For code generation, functional correctness receives priority over textual resemblance. For debugging, a repair counts as successful when the modified program passes the benchmark's relevant tests. For repository repair, success requires resolving the issue under the evaluation harness rather than merely generating a plausible patch. This distinction reduces the risk of interpreting fluent output as correct software. The analysis also considers security and developer interaction because a functionally correct snippet can still introduce vulnerabilities or impose additional verification work (Moradi Dakhel et al., 2023; Pearce et al., 2022; Vaithilingam et al., 2022).

Source quality was assessed through publication status, benchmark transparency, task reproducibility, and clarity of the reported metric. Numerical values were retained only when they could be traced directly to the original paper or benchmark report. When a source described a directional result without a stable cross-model score, the synthesis reports the direction rather than creating a numerical estimate. This procedure keeps the results anchored to published evidence and avoids combining incompatible measurements into a synthetic performance index.

The analysis also distinguishes model capability from evaluation scaffolding. Sampling count, execution tools, runtime feedback, prompt format, and repository-navigation mechanisms can substantially alter performance. These factors are treated as part of the evaluated system because a coding assistant in practice operates through an interface and toolchain. At the same time, comparisons are interpreted cautiously when two studies use different scaffolds or generation budgets.

4. Results

Table 1. Verified benchmark findings used in the comparative synthesis

Evidence	Task	Reported result
Chen et al. (2021)	HumanEval generation	Codex: 28.8% pass@1; 70.2% pass@100.
Li et al. (2022)	Competitive generation	AlphaCode averaged a top-54.3% participant ranking in simulated Codeforces contests.
Sobania et al. (2023)	QuixBugs repair	ChatGPT repaired 31/40 bugs when conversational hints were added.
Tian et al. (2024)	DebugBench	GPT-4: 75.0%; GPT-3.5: 62.1% overall bug-instance pass rate.
Jimenez et al. (2024)	SWE-bench	Claude 2 resolved 1.96% of real repository issues in the original evaluation.
Jain et al. (2025)	LiveCodeBench	Evaluation of 50+ models exposed contamination and saturation in older coding benchmarks.

4.1 Code-Generation Performance

The code-generation evidence indicates strong performance on bounded tasks but a substantial dependence on sampling and benchmark design. HumanEval shows that a single Codex sample solved 28.8% of tasks, whereas up to 100 samples raised the probability of at least one correct solution to 70.2% (Chen et al., 2021). The practical interpretation is not that repeated generation automatically creates reliable software; rather, LLMs can search a broad solution space when candidate programs are screened by tests. AlphaCode demonstrates the same principle at higher complexity: large-scale generation followed by filtering enabled competition-level performance that would not be represented by a single greedy completion (Li et al., 2022).

The comparison also suggests that conversational decomposition improves coding performance. CodeGen's multi-turn results support the idea that complex intentions can be separated into smaller subproblems, reducing the burden placed on one prompt (Nijkamp et al., 2023). This matters for practical programming because software requirements often contain several constraints that are easier to validate incrementally than in one generated block.

4.2 Debugging Performance

Debugging results are strongest when the defect is local and the feedback identifies the failing region. In DebugBench, GPT-4-0613 achieved a 75.0% overall pass rate and GPT-3.5-turbo-0613 achieved 62.1% (Tian et al., 2024). The detailed results show much higher success on several syntax cases than on complex logical or multiple-error cases. For example, GPT-4 repaired 93.6% of missing-colon instances but 55.9% of quadruple-bug instances. GPT-3.5 showed the same direction, scoring 80.9% and 38.7% on those categories, respectively. These values indicate that debugging quality falls as the repair requires deeper semantic reasoning and coordination of several changes.

Interactive feedback is beneficial but not uniformly. The QuixBugs study improved ChatGPT's repair success to 31 of 40 bugs after additional hints (Sobania et al., 2023). DebugBench similarly found that runtime messages helped syntax and reference defects, but they could reduce performance on logic errors (Tian et al., 2024). Error messages therefore function best as evidence within a reasoning process rather than as a complete explanation of program behavior.

4.3 Real-World Software Engineering

The most pronounced performance decline appears when the unit of analysis changes from a function to a repository. SWE-bench required models to work with issue descriptions, large code contexts, dependencies, multiple files, and repository tests. Claude 2 resolved 1.96% of the original benchmark issues (Jimenez et al., 2024), a result far below function-level benchmark scores. This difference is not contradictory; the benchmarks assess different levels of software-engineering competence. A repository patch can fail because of context selection, environment assumptions, test

interpretation, or unintended effects elsewhere in the codebase even when the model can generate a correct standalone function.

LiveCodeBench adds a second caution: benchmark familiarity can inflate apparent progress. By using newer contest problems and time-segmented evaluation, the benchmark detected contamination and overfitting patterns across open and closed models (Jain et al., 2025). Consequently, a credible evaluation of coding LLMs benefits from recent problems, executable tests, and task settings that resemble the intended deployment context.

5. Discussion

The comparative evidence supports a layered view of LLM programming capability. At the first layer, models are effective probabilistic synthesizers. They can produce candidate implementations rapidly and often solve concise tasks when specifications and tests are clear. At the second layer, models act as repair assistants. They perform well when the defect is syntactic, referential, or otherwise local, especially when execution feedback is available. At the third layer, real software maintenance demands persistent reasoning across repositories, dependencies, tools, and tests. Current performance becomes less stable at this level.

This pattern explains why developer experience can be positive even when benchmark reliability remains imperfect. Programmers gain value from fast drafts, explanations, alternative implementations, and repair suggestions, while retaining responsibility for validation. Vaithilingam et al. (2022) found that users often preferred Copilot because it provided useful starting points, although generated code could be difficult to understand, edit, and debug. Moradi Dakhel et al. (2023) similarly concluded that Copilot can function as an asset for experienced developers but can become a liability when users cannot diagnose incorrect suggestions.

Security strengthens the case for verification. Pearce et al. (2022) reported that approximately 40% of Copilot-generated programs in their security-focused scenarios were vulnerable. Functional test success alone therefore does not establish production readiness. A robust workflow combines generation with unit tests, static analysis, security checks, dependency review, and human code inspection. In debugging, the same principle applies: a patch that satisfies visible tests may still overfit the test suite or introduce hidden regressions.

The findings also have implications for benchmark interpretation. Pass@1 rewards first-attempt correctness, pass@k measures the value of multiple candidates, repair rate measures successful correction of defective programs, and SWE-bench issue resolution measures end-to-end software maintenance. Treating these values as directly interchangeable obscures the actual capability being tested. The rapid evolution of models also means that benchmark freshness matters. LiveCodeBench's contamination-aware design and SWE-bench's repository-based tasks point toward evaluations that are dynamic, executable, and closer to real development environments.

For practice, the evidence favors a staged AI-assisted development process. A model can generate an initial implementation, but acceptance should depend on automated tests, linting, type checking, static analysis, and security scanning. Debugging interactions benefit from concrete failure evidence such as test names, expected outputs, stack traces, and minimized reproductions, while logic defects often require broader descriptions of intended behavior. Repository-scale work further benefits from retrieval of only the relevant files and symbols, explicit tool access, and repeated execution after each patch. This workflow uses the speed of LLM generation without treating language-model confidence as a substitute for software verification.

The same staged approach has educational implications. Novice programmers can learn from explanations and alternative solutions, but they may also accept incorrect code because generated answers are fluent and authoritative in tone. Experienced developers are better positioned to recognize weak assumptions, incomplete edge-case handling, or inappropriate abstractions. As a result, the value of an LLM is partly conditional on the user's ability to inspect and test its output, a conclusion that aligns with both usability and code-quality studies (Moradi Dakhel et al., 2023; Vaithilingam et al., 2022).

Several limitations remain in this comparison. The models were evaluated at different times, with different prompting procedures, context windows, sampling budgets, and tool scaffolds. Some foundational benchmarks are also vulnerable to training-data exposure. For these reasons, the synthesis emphasizes within-benchmark findings and cross-benchmark patterns rather than a single numerical ranking. Even with these constraints, the evidence converges on a consistent conclusion: LLMs are useful programming partners, but reliability decreases as tasks become more contextual, stateful, and semantically complex.

6. Conclusion

Large language models have established meaningful capability in automated code generation and debugging. Published benchmarks show strong function-level synthesis, substantial gains from repeated sampling, and effective repair of many syntax and reference defects. The same evidence also identifies persistent weaknesses in logical debugging, multi-error repair, security, and repository-scale maintenance. The sharp gap between HumanEval-style tasks and SWE-bench-style issues demonstrates that software engineering competence cannot be inferred from a single coding benchmark. LLMs are therefore best treated as high-speed assistants embedded within test-driven and review-based workflows. Their greatest practical value arises when generated code is executed, checked, refined, and integrated under human supervision rather than accepted solely because it appears syntactically and semantically plausible.

References

- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q. V., & Sutton, C. (2021). Program synthesis with large language models. arXiv. <https://doi.org/10.48550/arXiv.2108.07732>
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., 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. arXiv. <https://doi.org/10.48550/arXiv.2107.03374>
- Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), Article 220. <https://doi.org/10.1145/3695988>
- Jain, N., Han, K., 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. In *The Thirteenth International Conference on Learning Representations*. https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html
- 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? In *The Twelfth International Conference on Learning Representations*. https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html
- Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Goyal, S., Cherepanov, A., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092-1097. <https://doi.org/10.1126/science.abq1158>
- Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (2023). GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, 203, 111734. <https://doi.org/10.1016/j.jss.2023.111734>
- Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*. <https://iclr.cc/virtual/2023/poster/11335>

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy (SP) (pp. 754-768). IEEE. <https://doi.org/10.1109/SP46214.2022.9833571>

Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023). An analysis of the automatic bug fixing performance of ChatGPT. In 2023 IEEE/ACM International Workshop on Automated Program Repair (APR) (pp. 23-30). IEEE. <https://doi.org/10.1109/APR59189.2023.00012>

Tian, R., Ye, Y., Qin, Y., Cong, X., Lin, Y., Pan, Y., Wu, Y., Hui, H., Liu, W., Liu, Z., & Sun, M. (2024). DebugBench: Evaluating debugging capability of large language models. In Findings of the Association for Computational Linguistics: ACL 2024 (pp. 4173-4198). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2024.findings-acl.247>

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In CHI Conference on Human Factors in Computing Systems Extended Abstracts. Association for Computing Machinery. <https://doi.org/10.1145/3491101.3519665>