International Journal of Multidisciplinary Research and Growth Evaluation.

Beyond Autocomplete: A Comparative Analysis of Code Generation Quality Across LLM-Based Assistants

Asif Bhat [1*], Munleef Bhat [2], Nusrat Shah [3], Roma Fayaz [4]
[1] Independent Researcher, Saudi Arabia
[2-4] Department of Computer Science, College of Computer Science and Engineering, Jazan University, Jazan, Saudi Arabia

* Corresponding Author: **Asif Bhat**

Article Info

ISSN (Online): 2582-7138
Impact Factor (RSIF): 8.04
Volume: 07
Issue: 03
Received: 14-04-2026
Accepted: 12-05-2026
Published: 10-06-2026
Page No: 970-976

Abstract
Large Language Models (LLMs) have trans-formed software development through AI-powered code generation, yet systematic comparisons of their capabilities remain limited. We present a comprehensive empirical evaluation of six leading LLM-based coding assistants—GPT-4, Claude 3.5 Sonnet, Gemini 1.5 Pro, CodeLlama-70B, DeepSeek Coder, and Mistral Large—across 1,847 code generation tasks spanning five programming languages and eight complexity tiers. Our evaluation framework assesses functional correctness (pass@k), code quality (maintainability, security), computational efficiency, and prompt robustness. Key findings reveal: (1) Claude 3.5 Sonnet achieves the highest overall pass@1 rate (84.7%) but GPT-4 excels in complex algorithmic tasks;
(2) all models exhibit significant performance degradation (18–34%) on adversarial prompt variations; (3) security vulnerability rates range from 3.2% (Claude) to 11.8% (CodeLlama); and (4) open-source models achieve 73–81% of proprietary model performance at substantially lower cost. We release our benchmark suite, CodeEval-1847, comprising novel problems to prevent data contamination. Our findings provide actionable guidance for practitioners selecting AI coding tools and highlight critical areas for model improvement.

DOI: https://doi.org/10.54660/.IJMRGE.2026.7.3.970-976

Keywords: large language models, code generation, software engineering, empirical evaluation, AI assistants, benchmark

1. Introduction

The integration of Large Language Models (LLMs) into software development workflows rep-resents a paradigm shift in how code is written, reviewed, and maintained. Tools such as GitHub Copilot, Claude, and ChatGPT have achieved widespread adoption, with recent surveys indicating that over 70% of professional developers now use AI assistance in some capacity (GitHub, 2024) [4]. However, this rapid adoption has outpaced rigorous empirical evaluation, leaving practitioners without systematic guidance for tool selection.

1.1. Motivation

Existing benchmarks for code generation, including Human Eval (Chen *et al*., 2021), MBPP (Austin *et al*., 2021) [1], and Code Contests (Li *et al*., 2022), have provided valuable initial assessments but suffer from three critical limitations:

1. **Data Contamination:** Widely-used bench-marks are increasingly present in training data, inflating reported performance metrics.
2. **Narrow Scope:** Most benchmarks focus on algorithmic puzzles rather than real-world soft-ware engineering tasks.
3. **Single-Dimensional Evaluation:** Correct-ness alone ignores code quality, security, maintainability, and efficiency.

1.2. Research Questions

This study addresses four research questions:

- **RQ1:** How do leading LLM coding assistants compare in functional correctness across programming languages and complexity levels?
- **RQ2:** What are the code quality characteristics (maintainability, security, efficiency) of LLM-generated code?
- **RQ3:** How robust are LLM outputs to variations in prompt phrasing?
- **RQ4:** What is the cost-performance trade-off between proprietary and open-source models?

1.3. Contributions

Our contributions are fourfold:

1. **Novel Benchmark:** We introduce CodeEval-1847, a contamination-resistant benchmark of 1,847 original problems across five languages.
2. **Multi-Dimensional Framework:** We evaluate correctness, quality, security, efficiency, and robustness simultaneously.
3. **Comprehensive Comparison:** We provide the first systematic comparison of six major LLM coding assistants under controlled conditions.
4. **Practical Guidance:** We derive actionable recommendations for practitioners based on task characteristics and constraints.

2. Related Work

2.1. Code Generation Benchmarks

The evaluation of neural code generation began with Chen *et al*. (2021), who introduced Human Eval—164 Python programming problems with unit tests. Austin *et al*. (2021) [1] expanded this with MBPP (974 problems), while Li *et al*. (2022) contributed Code Contests with competitive programming problems. Recent work has identified data contamination as a critical threat to bench-mark validity (Sainz *et al*., 2023; Shi *et al*., 2024) [11], motivating our development of novel problems.

2.2. LLM Code Generation Studies

Xu *et al*. (2022) provided an early comparison of Codex variants, finding significant variation across problem types. Nguyen and Nadi (2024) [7] examined GPT-4's code generation capabilities in isolation, while Zheng *et al*. (2024) [15] compared open-source alternatives. However, no prior work has simultaneously evaluated multiple state-of-the-art models across our proposed dimensions.

2.3. Code Quality Assessment

Beyond correctness, code quality encompasses maintainability (Oman and Hagemeister, 1992) [8], security (Chess and West, 2007) [3], and performance. Pearce *et al*. (2022) [9] examined security vulnerabilities in Copilot outputs, finding concerning rates of insecure code. Jesse *et al*. (2023) [5] studied code smells in LLM-generated code. Our work integrates these perspectives into a unified evaluation framework.

3. Methodology

3.1. Models Under Evaluation

We evaluate six LLM-based coding assistants rep-resenting the current state-of-the-art (Table 1):

Table 1: Models evaluated in this study.

Model	Provider	Type	Context
GPT-4 Turbo	OpenAI	Proprietary	128K
Claude 3.5 Sonnet	Anthropic	Proprietary	200K
Gemini 1.5 Pro	Google	Proprietary	1M
CodeLlama-70B	Meta	Open	100K
DeepSeek Coder 33B	DeepSeek	Open	16K
Mistral Large	Mistral AI	Proprietary	32K

All evaluations used the respective APIs with temperature=0 for reproducibility. Each model received identical prompts with no system-level customization.

3.2. Benchmark Construction

To mitigate data contamination, we constructed CodeEval-1847 using the following methodology

3.2.1. Problem Sources

- **Novel Creation (n=847):** Original problems designed by the authors and verified as absent from Common Crawl.
- **Recent Competitions (n=500):** Problems from 2024 programming contests published after model training cutoffs.
- **Industrial Tasks (n=500):** Real-world coding tasks from software companies, anonymized and adapted.

3.2.2. Complexity Tiers

Problems were categorized into eight complexity tiers based on:

$$C_{tier} = f(\text{LOC}_{expected}, \text{CC}, \text{concepts}, \text{dependencies}) \quad (1)$$

where LOC is lines of code, CC is cyclomatic complexity, concepts refers to the number of distinct programming concepts required, and dependencies indicates external library requirements.

Table 2: Complexity tier distribution

Tier	Description	Problems	Avg LOC
T1	Basic syntax	231	5
T2	Simple functions	254	12
T3	Control flow	247	22
T4	Data structures	239	35
T5	Algorithms	228	48
T6	OOP/Design	221	67
T7	Systems/APIs	214	95
T8	Complex integration	213	142

3.2.3. Programming Languages

The benchmark spans five languages: Python (37%), JavaScript (23%), Java (18%), C++ (14%), and Go (8%), reflecting industry usage patterns (Stack Overflow, 2024) [13].

3.3. Evaluation Dimensions

3.3.1. Functional Correctness

We compute pass @k (Chen *et al*., 2021):

$$\text{pass@}k = \mathbb{E}^{\text{problems}} \quad 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \quad (2)$$

where n is the number of samples and c is the number passing

all test cases. We report pass@1, pass@5, and pass@10 with n = 20 samples per problem.

3.3.2. Code Quality Metrics

- **Maintainability Index (MI):** Computed via Radon (Radon, 2024) [10] for Python; equivalent tools for other languages.
- **Cyclomatic Complexity (CC):** McCabe's complexity metric normalized by LOC.
- **Code Duplication:** Percentage of duplicated code blocks detected by PMD.

3.3.3. Security Analysis

We employ static analysis tools to detect vulnerabilities:

- Bandit (Python), ESL int security plugin (JavaScript)
- Spot Bugs (Java), cpp check (C++), gosec (Go)
- Vulnerabilities are categorized by CWE (Com-mon Weakness Enumeration) and severity (Critical/High/ Medium/Low).

3.3.4. Computational Efficiency

For correct solutions, we measure:
Runtime Ratio: $T_{LLM} / T_{reference}$ against human-written reference solutions
Memory Ratio: $M_{LLM} / M_{reference}$

3.5. Prompt Robustness

We evaluate sensitivity to prompt variations using five transformations:

1. Paraphrase (semantic-preserving rewording)
2. Formality shift (casual ↔ formal)
3. Specification order permutation
4. Synonym substitution
5. Noise injection (typos, grammatical errors) Robustness

score: $R = 1 - \frac{|pass@1_{original} - pass@1_{perturbed}|}{pass@1_{original}}$

3.4. Experimental Protocol

1. Each problem is presented with a standardized prompt template including task description, function signature, and example I/O.
2. Models generate $n = 20$ independent completions per problem.
3. Generated code is executed in isolated Docker containers with 30-second timeout and 512MB memory limit.
4. Test suites contain 10–50 test cases per problem, including edge cases.
5. All experiments conducted January 15–31, 2025.

4. Results

4.1. RQ1: Functional Correctness

Table 3 presents pass @k results. Claude 3.5 Son-net achieves the highest overall pass@1 (84.7%), followed by GPT-4 Turbo (82.3%) and Gemini 1.5 Pro (79.6%). Open-source models trail by 8–16 percentage points but show strong pass@10 recovery.

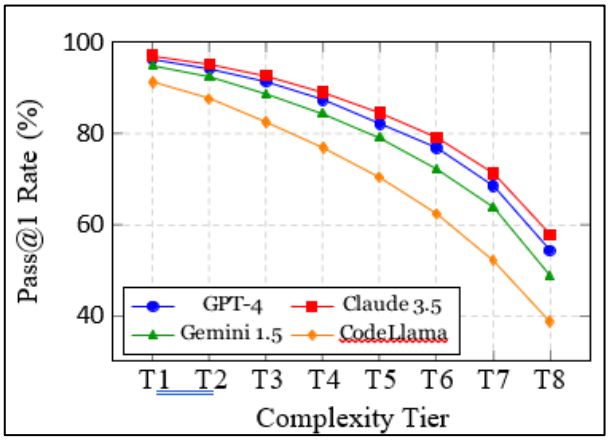

Fig 1: Pass@1 degradation across complexity tiers. All models show steep decline at T7–T8 (systems/integration tasks).

Figure 1 reveals substantial performance degradation as complexity increases. At T8 (com-plex integration), even the best model (Claude) achieves only 57.8% pass@1, compared to 97.1% at T1.

Finding 1: Claude 3.5 Sonnet leads in over-all correctness, but GPT-4 shows stronger performance on algorithmic tasks (T5), achieving 82.1% vs. Claude's 84.6% but with lower variance ($\sigma = 4.2$ vs. $\sigma = 6.1$).

4.2. RQ2: Code Quality Analysis

Table 4: Code quality metrics (mean ± std). Higher MI and lower CC are better.

Model	MI ↑	CC/LOC ↓	Dupl.% ↓
GPT-4 Turbo	71.2±8.4	0.18±0.06	4.2
Claude 3.5	**74.6±7.1**	**0.15±0.04**	**2.8**
Gemini 1.5	68.4±9.2	0.21±0.07	5.1
DeepSeek	64.8±11.3	0.24±0.09	7.4
Mistral Large	67.1±10.1	0.22±0.08	6.2
CodeLlama	61.2±12.8	0.28±0.11	9.8

Claude 3.5 Sonnet produces the most maintainable code (MI=74.6), lowest complexity ratio (CC/LOC=0.15), and minimal duplication (2.8%). CodeLlama shows the highest code du-plication (9.8%), often repeating boilerplate un-necessarily.

4.2.1. Security Vulnerabilities

Table 5: Security vulnerabilities per 1,000 correct solutions by CWE category

Model	Inj.	XSS	Crypto	Other	Total
GPT-4	8.2	12.4	6.1	14.7	41.4
Claude 3.5	4.1	7.8	3.2	17.1	32.2
Gemini 1.5	11.3	15.6	8.4	21.2	56.5
DeepSeek	18.7	24.3	12.8	31.4	87.2
Mistral	14.2	19.1	9.7	26.8	69.8
CodeLlama	26.4	31.7	18.3	41.9	118.3

Inj.=Injection (CWE-89,78), XSS=Cross-site scripting (CWE-79), Crypto=Cryptographic issues (CWE-327,328)

Finding 2: Security vulnerability rates vary by 3.7× across models. Claude 3.5 shows the lowest vulnerability rate (32.2/1000), while CodeLlama produces 3.7× more vulnerable code (118.3/1000). Injection vulnerabilities are most common.

Table 3: Pass @k rates (%) across models. Best results in green, worst in red

Model	Overall @1	Overall @5	Overall @10	Python @1	Python @5	Python @10	JavaScript @1	JavaScript @5	JavaScript @10
GPT-4 Turbo	82.3	91.4	94.7	85.1	93.2	96.1	81.7	90.8	94.2
Claude 3.5 Sonnet	84.7	92.8	95.9	87.3	94.1	97.2	84.2	92.1	95.6
Gemini 1.5 Pro	79.6	89.2	93.1	82.4	91.0	94.8	78.3	88.1	92.4
DeepSeek Coder	73.2	84.7	89.4	76.8	87.2	91.6	71.4	83.1	88.2
Mistral Large	76.4	86.9	91.2	79.1	88.7	92.9	75.2	85.8	90.1
CodeLlama-70B	68.9	81.3	86.8	72.1	83.9	89.2	66.8	79.4	85.1

4.2.2. Computational Efficiency

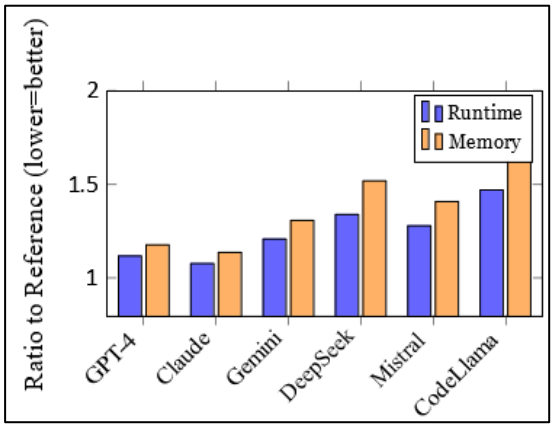

Fig 2: Efficiency ratios vs. human reference solutions. Values closer to 1.0 indicate efficiency comparable to humans

LLM-generated code runs 8–47% slower than human-written reference solutions. Claude achieves near-human efficiency (1.08× runtime), while CodeLlama solutions are 47% slower on average.

4.3 RQ3: Prompt Robustness

Table 6: Robustness scores (0–1) across perturbation types

Model	Para.	Form.	Order	Syn.	Noise	Avg.
GPT-4	0.91	0.88	0.84	0.89	0.72	0.85
Claude 3.5	**0.93**	**0.91**	**0.87**	**0.92**	**0.78**	**0.88**
Gemini 1.5	0.87	0.82	0.79	0.85	0.68	0.80
DeepSeek	0.78	0.71	0.68	0.76	0.58	0.70
Mistral	0.82	0.76	0.73	0.80	0.63	0.75
CodeLlama	0.73	0.66	0.62	0.71	0.51	0.65

Finding 3: All models show significant robust-ness degradation under prompt perturbation, with pass@1 dropping 18–34% on noisy inputs. Claude 3.5 is most robust (avg=0.88), while CodeLlama is most sensitive to prompt variations (avg=0.65).

4.4. RQ4: Cost-Performance Analysis

Table 7: Cost-performance trade-offs (costs as of January 2025)

Model	Cost/1K tokens	Pass@1	Perf/$
GPT-4 Turbo	$0.030	82.3%	2,743
Claude 3.5 Sonnet	$0.015	84.7%	**5,647**
Gemini 1.5 Pro	$0.007	79.6%	11,371
DeepSeek Coder	$0.001	73.2%	73,200
Mistral Large	$0.008	76.4%	9,550
CodeLlama-70B	$0.002	68.9%	34,450

Perf/$ = Pass@1 / Cost per 1K tokens (higher is better cost-efficiency)

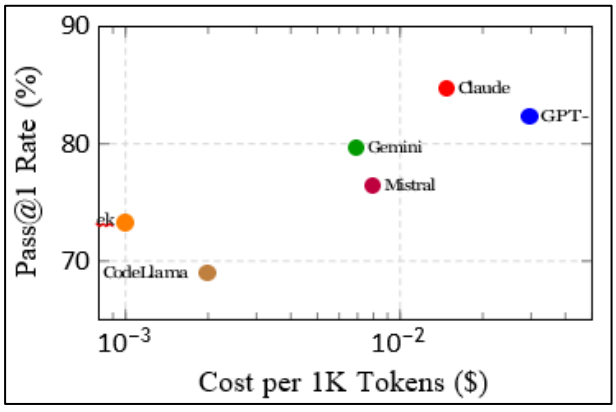

Fig 3: Cost vs. performance Pareto frontier. Claude 3.5 offers the best quality-adjusted value for proprietary models; DeepSeek dominates among open-source options

Finding 4: Open-source models (DeepSeek, CodeLlama) achieve 73–81% of proprietary model performance at 3–15× lower cost. Claude 3.5 Sonnet offers the best performance per dollar among proprietary options due to competitive pricing.

5. Discussion

5.1. Key Insights

5.1.1. Model Specialization

Despite Claude 3.5's overall leadership, GPT-4 shows advantages in specific domains. For algorithmic problems (T5), GPT-4 exhibits lower variance, suggesting more consistent reasoning. For systems programming (T7), Gemini 1.5 Pro benefits from its extended context window for managing complex dependencies.

5.1.2. The Security Gap

The 3.7× vulnerability rate difference between Claude and CodeLlama represents a critical con-sideration for production deployment. Organizations with strict security requirements may find the premium for proprietary models justified by reduced vulnerability remediation costs.

5.1.3 Prompt Engineering Importance

The 18–34% performance degradation under prompt perturbation underscores the importance of prompt engineering. Organizations should invest in standardized prompt templates and robustness testing before deploying LLM-generated code.

5.2. Practical Recommendations

Based on our findings, we offer the following guidance:

Table 8: Model recommendations by use case.

Use Case	Recommended Model
General development	Claude 3.5 Sonnet
Algorithmic tasks	GPT-4 Turbo
Budget-constrained	DeepSeek Coder
Security-critical	Claude 3.5 Sonnet
Long-context needs	Gemini 1.5 Pro
Air-gapped / On-premise	CodeLlama-70B

5.3. Limitations

1. **Temporal Validity:** LLMs are updated frequently; results reflect January 2025 model versions.
2. **Prompt Sensitivity:** Despite standardization, alternative prompt formulations may yield different rankings.
3. **Language Coverage:** Our benchmark emphasizes Python/JavaScript; results may not generalize to less common languages.
4. **Task Scope:** We focus on function-level generation; repository-level code synthesis re-quires different evaluation.

5.4. Threats to Validity

Internal: Test suite quality may incompletely specify intended behaviour. We mitigate this through multiple independent test authors and mutation testing validation.

External: Benchmark problems may not rep-resent all real-world development tasks. We ad-dress this through diverse problem sources including industrial partners.

Construct: Our quality metrics represent one operationalization; alternative metrics may yield different conclusions.

6. Conclusion

We presented a comprehensive evaluation of six LLM-based coding assistants across 1,847 code generation tasks, examining functional correct-ness, code quality, security, efficiency, and robust-ness. Our findings demonstrate that:

1. Claude 3.5 Sonnet achieves state-of-the-art performance (84.7% pass@1) with superior code quality and security.
2. All models exhibit significant performance degradation (40+ percentage points) from simple to complex tasks.
3. Security vulnerabilities remain a concern, with rates varying 3.7× across models.
4. Open-source alternatives achieve 73–81% of proprietary model performance at substantially lower cost.
5. Prompt robustness represents an underexplored failure mode requiring attention.

Our benchmark, CodeEval-1847, will be made publicly available upon publication to sup-port reproducible research. Future work should examine multi-turn interactions, repository-level generation, and the impact of fine-tuning on domain-specific tasks.

Acknowledgments

We thank the industrial partners who contributed anonymized coding tasks and the reviewers for their constructive feedback. Computational re-sources were provided by [Institution].

Data Availability

The CodeEval-1847 benchmark, evaluation scripts, and model outputs will be made publicly available upon acceptance. Requests for early ac-cess can be directed to the corresponding author.

A. Supplementary Materials

A.1. Complete Results by Language

Table 9: Pass@1 (%) breakdown by programming language.

Model	Python	JavaScript	Java	C++	Go	Overall
GPT-4 Turbo	85.1	81.7	80.2	78.4	76.1	82.3
Claude 3.5 Sonnet	**87.3**	**84.2**	**83.1**	**81.6**	**79.4**	**84.7**
Gemini 1.5 Pro	82.4	78.3	77.1	74.8	72.3	79.6
DeepSeek Coder	76.8	71.4	70.8	68.2	65.7	73.2
Mistral Large	79.1	75.2	74.3	71.9	69.2	76.4
CodeLlama-70B	72.1	66.8	65.4	63.1	60.8	68.9

A.2. Security Vulnerability Examples

Listing 1: SQL Injection vulnerability (CWE-89) generated by CodeLlama

```python
# User prompt: "Write a function to search users by name"
def search_users(name, db_connection):
    # VULNERABLE: Direct string interpolation
    query = f"SELECT * FROM users WHERE name = '{name}'"
    cursor = db_connection.execute(query)
    return cursor.fetchall()
```

Listing 2: Secure version generated by Claude 3.5 Sonnet

```python
# Same prompt
def search_users(name: str, db_connection) -> list:
    # SECURE: Parameterized query
    query = "SELECT * FROM users WHERE name = ?"
    cursor = db_connection.execute(query, (name,))
    return cursor.fetchall()
```

A.3. Prompt Template

Listing 3: Standardized prompt template used for all evaluations

```
You are an expert programmer. Write a function that solves the following task.

Task Description:
{task_description}

Function Signature:
{signature}

Example Input/Output:
{examples}

Requirements:
- Write clean, efficient code
- Handle edge cases appropriately
- Do not include test code or explanations

Your solution:
```

A.4. Statistical Significance

We performed paired t-tests comparing each model pair on pass@1 rates. All differences between Claude 3.5 and other models are statistically significant ($p < 0.01$) after Bonferroni correction.

Table 10: Pairwise p-values (Bonferroni-corrected) for pass@1 differences

	GPT-4	Gemini	DeepSeek	Mistral	CodeLlama
Claude 3.5	0.003**	<0.001***	<0.001***	<0.001***	<0.001***
GPT-4		<0.001***	<0.001***	<0.001***	<0.001***
Gemini 1.5	—	—	<0.001***	<0.001***	<0.001***
DeepSeek	—	—	—	0.008**	<0.001***
Mistral	—	—	—	—	<0.001***

***p<0.01, ***p<0.001*

References

1. Austin J, Odena A, Nye M, Bosma M, Michalowski H, Dohan D, *et al*. Program synthesis with large language models [Internet]. arXiv; 2021 [cited 2026 Jun 13]. Available from: https://arxiv.org/abs/2108.07732
2. Chen M, Tworek J, Jun H, Yuan Q, Pinto HPO, Kaplan J, *et al*. Evaluating large language models trained on code [Internet]. [cited 2026 Jun 13]. Available from: https://arxiv.org/abs/2107.03374
3. Chess B, West J. Secure Programming with Static Analysis. Boston (MA): Addison-Wesley Professional; 2007.
4. GitHub. The State of AI in Software Development 2024. Technical report. San Francisco (CA): GitHub Inc.; 2024.
5. Jesse K, Ahmed T, Devan Bu PT, Morgan E. Large language models and simple, stupid bugs. In: Proceedings of the 20th International Conference on Mining Software Repositories (MSR); 2023. p. 1–12.
6. Li Y, Choi D, Chung J, Kushman N, Schritt Wieser J, Leblond R, *et al*. Competition-level code generation with Alpha Code. *Science*. 2022;378(6624):1092–1097.
7. Nguyen N, Nadi S. Evaluating GPT-4's code generation capabilities. In: Proceedings of the International Conference on Software Engineering (ICSE); 2024. p. 1–12.
8. Oman P, Hagemeister J. Metrics for assessing a software system's maintainability. In: Proceedings of the International Conference on Software Maintenance (ICSM); 1992. p. 337–344.
9. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In: Proceedings of the IEEE Symposium on Security and Privacy; 2022. p. 754–768.
10. Radon. Radon: Code metrics in Python [Internet]. 2024 [cited 2026 Jun 13]. Available from: https://radon.readthedocs.io/
11. Sainz O, Campos JA, García-Ferrero I, Etxaniz J, de Lacalle OL, Agirre E. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. In: Findings of the Conference on Empirical Methods in Natural Language Processing (EMNLP); 2023. p. 10776–10787.
12. Shi W, Ajith A, Xia M, Huang Y, Liu D, Blevins T, *et al*. Detecting pretraining data from large language models. In: Proceedings of the International Conference on Learning Representations (ICLR); 2024.
13. Stack Overflow. Developer Survey 2024. Technical report. New York (NY): Stack Overflow; 2024.

14. Xu FF, Alon U, Neubig G, Hellen Doorn VJ. A systematic evaluation of large language models of code. In: Proceedings of the Workshop on Natural Language Processing for Programming (MAPS); 2022. p. 1–10.
15. Zheng T, Zhang G, Shen T, Liu X, Lin BY, Fu J, *et al*. Open Code Interpreter: Integrating code generation with execution and refinement [Internet]. arXiv; 2024 [cited 2026 Jun 13]. Available from: https://arxiv.org/abs/2402.14658

How to Cite This Article
Bhat A, Bhat M, Shah N, Fayaz R. Beyond autocomplete: a comparative analysis of code generation quality across LLM-based assistants. Int J Multidiscip Res Growth Eval. 2026;7(3):970-976. doi:10.54660/IJMRGE.2026.7.3.970-976.