STM JOURNALS

Recent Trends in
Programming Languages

ISSN: 2455-1821
Volume 12, Issue 3, 2025
September–December
DOI (Journal): 10.37591/RTPL

https://journals.stmjournals.com/rtpl

Review RTPL

Methodology for Evaluating Code Synthesis in Large Language Models: ChatGPT and Copilot: A Review

Saurabh Sheoran[1,*], Dinesh Kumar[2]

Abstract

The authors introduce a comprehensive framework to assess the code-generation capabilities of large language models, focusing on ChatGPT and Copilot through a benchmark suite of 25 program synthesis tasks. Their main goal was to show why making proper comparisons is important, they did not focus on choosing the newest models, since they keep changing frequently. The critique examines how the methodology addresses both functional and non-functional aspects of code. In the functional testing domain, ChatGPT delivered 17 fully correct solutions, compared to Copilot's 13 flawless outputs. Non-functional assessment, meanwhile, revealed that, despite overall decent quality, both models still exhibited recognizable code smells. Further, the study incorporated human evaluators to judge code quality, offering nuanced insights into each LLM's strengths and weaknesses. The critique emphasizes the practical significance of these findings in helping developers choose the most appropriate LLM for their coding needs and highlights how systematic evaluation can guide tool selection in real-world development scenarios. LLM can be very useful in software development field for software developers. Finally, it advocates for expanding the evaluation lens to include more parameters in result, use of other LLMs and programming languages for evaluation, and newer model iterations to gain a more holistic understanding of LLMs' code-synthesis abilities.

Keywords: Code generator, LLM, ChatGPT, CoPilot, software development

INTRODUCTION

Large language models like ChatGPT and Copilot are changing software development by converting plain-language prompts into functional code. These AI assistants, often embedded in popular IDEs, can auto complete code snippets, transform descriptive comments into implementations, and help generate documentation. While developers are depending on them more than ever, selecting the most suitable LLM for coding tasks remains a challenging decision. Existing studies often evaluate models based on functional correctness whether the generated code compiles or passes a limited set of tests.

Ságodi *et al.* proposed an evaluation methodology that assesses both functional and non-functional quality of code synthesized by LLMs [1]. In their study, they apply this methodology to a benchmark of 25 tasks [2], comparing ChatGPT and Copilot. Their results show that ChatGPT achieved 17 fully correct solutions versus Copilot's 13. They also performed static code analysis to detect patterns and code smells, finding both tools generated high-quality code overall, yet each exhibited its own stylistic issues. Human reviewers confirmed these findings, validating the methodology.

***Author for Correspondence**
Saurabh Sheoran
E-mail: saurabhsheoran44@gmail.com

[1]Student, Department of Computer Science and Engineering, BRCM College of Engineering and Technology, Bahal, Bhiwani, Haryana, India
[2]Professor, Department of Computer Science and Engineering, BRCM College of Engineering and Technology, Bahal, Bhiwani, Haryana, India

Received Date: April 11, 2025
Accepted Date: June 28, 2025
Published Date: October 17, 2025

Citation: Saurabh Sheoran, Dinesh Kumar. Methodology for Evaluating Code Synthesis in Large Language Models: ChatGPT and Copilot: A Review. Recent Trends in Programming Languages. 2025; 12(3): 1–7p.

ChatGPT is a chatbot application built by OpenAI that can process image(s), text, generate code and audio inputs. GitHub's Copilot functions as an AI-powered coding assistant that aids developers by offering contextual code suggestions, from single lines to entire functions, helping to streamline tasks like writing, searching, and brainstorming code. It can also automate repetitive tasks [3, 4].

They can do things like:

- People can do text-based conversation with LLM. It can answer on any topic.
- Translate natural language to programming code.
- It can create graphs and charts depending on our inputs.
- It can analyze data.

Advantages of ChatGPT/Copilot/other LLMs:

- LLMs can seamlessly translate content between languages, providing instant multilingual support.
- They also serve as valuable assistive tools for students with learning disabilities.
- It provides responses to queries in very quick time.
- Capable of converting everyday/natural language descriptions directly into executable programming code.

Disadvantages of ChatGPT/Copilot/other LLMs:

- It is possible that it does not provide exact or accurate information.
- These models have been trained with lot of information, but still in growing world of information, they lack in knowledge.
- They can simulate natural conversation, but it does not possess the emotional intelligence found in human interactions.

LITERATURE REVIEW

Ságodi *et al*. provide an assessment of two widely used LLMs, namely Copilot and ChatGPT, by evaluating the quality of code they generate in Java and C++ [1]. This evaluation is based on the Natural Language Processing (NLP) capabilities of these models. Helmuth and Kelly introduce 25 new general program synthesis benchmark problems, forming the Program Synthesis Benchmark Suite 2 [2]. This new suite features problems gathered from diverse sources, such as programming katas and college courses. Black proposed the integration of static analysis into the process of ethical software development [5]. Since security must be built in from the start, static analysis should be conducted early in the development process to minimize vulnerabilities. Kaner and Bond developed a framework to evaluate proposed metrics and demonstrated its application through two examples that analyzed bug count data [6].

Vaithilingam *et al.* conducted a within-subjects user study involving 24 participants to explore how programmers interact with and perceive Copilot [7], a code generation tool powered by large language models (LLMs) [8]. They investigated the possibility of automatically generating fix recommendations for typical warnings generated by static code analysis tools. Marcilio *et al*. conducted a comprehensive, multi-method investigation into how developers utilize SonarQube, a popular static analysis platform [9–11]. Maddison *et al*. [12] presented a family of generative models for natural source code (NSC) that exhibit two main characteristics: first, they integrate both sequential and hierarchical structures; and second, they can closely interact with a compiler, enabling the use of compiler logic and abstractions to incorporate structure into the model. Sobania *et al.* assessed GitHub Copilot's efficacy on established program synthesis benchmarks and compared its outcomes with findings previously reported in the genetic programming domain [13].

Jain *et al*. proposed a method to enhance large language models with post-processing steps that utilize program analysis and synthesis techniques, enabling them to understand the syntax and semantics of

programs [14]. The study by Pearce *et al*. focuses on the challenges of designing prompts that effectively guide LLMs to produce corrected versions of insecure code [15, 16]. White *et al*. [17] introduce design techniques for software engineering through prompt patterns, aimed at addressing common challenges encountered when utilizing large language models (LLMs) to automate typical software engineering tasks.

METHODOLOGY

Ságodi *et al*. introduced an evaluation framework for assessing large language models (LLMs) in code synthesis, demonstrated through a comparison of ChatGPT and Copilot [1]. The methodology used a benchmark suite of 25 diverse programming tasks, which ensured a broad testbed across difficulty levels [2].

This study provides an in-depth assessment of the 2024 IEEE Access article by Ságodi *et al.*, called "Methodology for Code Synthesis Evaluation of LLMs Presented by a Case Study of ChatGPT and Copilot", and this article is published/shared under a Creative Commons Attribution-Non-Commercial-No Derivatives (CC BY-NC-ND) license (https://creativecommons.org/licenses/by-nc-nd/4.0/) [1]. The 25 tasks from the PSB2 suite were used in evaluation [1, 2]. The PSB2 suite, originally described by Helmuth and Kelly [2], is published under a Creative Commons Attribution-Non-Commercial-No Derivatives (CC BY-NC-ND 4.0) license (https://creativecommons.org/licenses/by-nc-nd/4.0/). Ságodi *et al*. describe below mentioned phases: Use of right prompt, functional assessment of code, technical assessment of code and human Judgment of code [1].

Use of Right Prompt

They mentioned that prompt provided for the models must be carefully selected, which presents a challenge as none of the compared models should be favored by optimized prompting. The difficulty and level of detail of the problems' descriptions are the main factors to consider when working with NLP-based systems.

Functionality Assessment of Code

Ensuring that code functions correctly is important in code generation. In the study by Ságodi *et al*. [1], tests were used to check if the generated code produced the expected results or not. ChatGPT achieved a higher rate of functional correctness, delivering 17 perfect solutions out of 25 tasks [2], while Copilot provided 13. This indicates that ChatGPT might have a stronger grasp of programming logic and problem solving.

Non-Functional/Technical Assessment of Code

When writing code, it is important not just that it works, but also that it is easy to read, maintain, and follows coding standards. Researchers looked at these non-functional aspects of AI-generated code and found "code smells"-signs as there might be problems affecting code quality. Both AI models produced code with noticeable code smells, pointing out areas where the AI-generated code could be improved to align with human coding standards.

Human Judgment of Code

To validate the automated findings, human reviewers manually examined samples of the code generated by both models [1]. This step provided qualitative insights into the models' performance, offering a human perspective on the usability and quality of the generated code. Developers were asked to rate the code on properties such as readability, usability, modifiability etc.

Some other actions or sub-phase or metrics:

1. *Task Selection:* To test the models, a diverse set of coding tasks were used with some actions like:
    - Code generation for solving problems/tasks.

- Code completion to fill in partially written code.
- Bug fixing in existing code.

2. *Performance Metrics:* To assess the models' outputs, they checked properties of generated code; some of them are mentioned below:
   - *Readability:* How easy it is for someone to read and understand the code.
   - *Usability:* How easy it is to use the code correctly, such as clear functions and good documentation.
   - *Modifiability:* How easy it is to change or update the code without breaking it.

ANALYSIS AND DISCUSSION

In the case study, the paper by Ságodi *et al*. compared the code synthesis capabilities of ChatGPT and Copilot based on a series of tasks [1]. For same, they have followed these steps, i.e. use of right prompt, functionality assessment, technical assessment and human judgment of code. Their main goal was to explain the importance of proper comparisons, they did not focus on improving prompts or picking the latest models, as those change frequently [1].

Experiment Setup

To demonstrate the application of methodology, considered factors like the models to compare and the programming language for evaluation. The following paragraphs explain the selection of these models and the reasoning behind choosing the programming language.

The comparison involved using ChatGPT and Copilot to generate code based on the same problem description. Copilot generates code snippets as developers write, while ChatGPT can modify the code based on further specifications from the developer. To ensure a fair comparison, the same problem description was provided to both models, and no further interaction was made beyond the initial code generation.

Chose C++ and Java programming languages [1]. Due to limited manpower, the team chose languages widely used in academia and industry. C++ was selected for its flexibility, supporting both object-oriented and non-object-oriented programming styles. Java was chosen as a language that handles memory automatically and primarily supports object-oriented programming.

In future study, some other LLMs, trending programming languages like Python and other can be used to test and evaluate large language models (LLMs), giving developers more options when choosing and working with LLMs.

Use of Right Prompt

In the methodology, the initial step involves selecting prompts with appropriate detail [1]. Given the complexity of this task, utilized the Program Synthesis Benchmark Suite (PSB2) [2], which comprises 25 programming tasks from diverse sources. They formatted these tasks as prompts by adding a consistent prefix, such as "Generate a C++ code to solve {task description}" or "Generate a Java code to solve {task description}". For this case study, they chose straightforward prompts to demonstrate the methodology's application [1].

Functionality Assessment of code

To assess the program's functional validity, they utilized the benchmark's suite input and output values for each task, incorporating edge cases, tests that challenge the boundaries of input values and random cases [2]. The number of edge cases varied from 5 to 40, depending on the specific task. Executed various edge case tests to ensure comprehensive evaluation [1]. Compared floating-point values in output using an epsilon value, where differences smaller than the specified epsilon were not considered as differing values.

Table 1. List of 25 programming tasks name of benchmark suite PSB2.

S.N.	Tasks name	S.N.	Tasks name	S.N.	Tasks name	S.N.	Tasks name	S.N.	Tasks name
1	Bouncing Balls	6	Camel Case	11	Cut Vector	16	Dice Game	21	Snow Day
2	Fizz Buzz	7	GCD	12	Leaders	17	Luhn	22	Twitter
3	Middle Character	8	Shopping List	13	Solve Boolean	18	Spin Words	23	Find Pair
4	Substitution Cipher	9	Vector Distance	14	Coin Sums	19	Fuel Cost	24	Mastermind
5	Basement	10	Bowling	15	Indices of Substring	20	Paired Digits	25	Square Digits

Here, the benchmark suite comprising 25 tasks to evaluate the code synthesis capabilities of ChatGPT and Copilot is shown in Table 1 [2].

For all 25 tasks, compared ChatGPT and CoPilot outputs for edge-exact, edge-epsilon, random-exact and random-epsilon scenarios [1].

Technical Assessment of Code

To conduct a technical quality assessment, employed static analysis tools for straightforward comparison. Utilized SonarQube and SonarScanner to analyze the projects, enhancing SonarQube's capabilities with the SourceMeter plug-in, which offers additional metrics, coding rules [1].

Used several metrics like Logical Lines of Code, Number of Statements, McCabe cyclomatic Complexity and Nesting Level (NLE), to evaluate the results of the static analysis. For all the 25 tasks [2], compared aforesaid metrics i.e. Logical Lines of Code (LLOC), Number of Statements (NOS) etc. for both languages i.e. C++ and Java. Then based on the result, compared both large language models i.e. ChatGPT and CoPilot.

In future study, static code analysis tools like Checkstyle, Pylint and others can be used depending on the programming language being used or as per user choice [3–8]. The technical evaluation can be further enhanced by including additional parameters in the results section, such as 'code duplication exists' etc. This will help broaden the assessment criteria and provide a more comprehensive evaluation of code quality.

Human Judgment of Code

During the inspection, developers were instructed to read the task descriptions that informed the generation of the source code [9–12]. After comprehending the task, examined the source codes produced by both models, presented side by side with their order randomized to eliminate bias regarding the models used. Developers then evaluated each source code on a scale. This scale was selected to prevent neutral responses, with positive values indicating approval and negative values indicating disapproval. The properties (evaluation parameters) assessed were: initial impression of code, readability of code, usability of code, modifiability of code and acceptance of code. In the human judgment phase/section, one can also include evaluation parameters like 'code comments present' in the results section separately with other evaluation parameters [13–17].

Combining human assessment with technical validation ensures high-quality code by merging contextual understanding with automated accuracy. Humans catch logic flaws and enhance maintainability, while tools enforce standards and detect common errors efficiently. This dual approach improves code reliability and team learning.

CONCLUSION AND FUTURE DIRECTIONS

This critique highlights method for evaluating the code generation abilities of LLMs, emphasizing the importance of detailed prompting, technical quality, human evaluation of code and functional testing of code generated by Large Language Models (LLMs), focusing on ChatGPT and Copilot. The case

study examines both models i.e. ChatGPT and Copilot, offering developers insights into their effective use in real-world software development. The work addresses two critical challenges:

- *Benchmark selection with manual assessment*: Previous evaluations often lack systematic methods for ensuring code meets developers' real-world quality and usability standards. By combining automated and human analyses, the proposed methodology fills this gap.
- *Model comparison*: With multiple LLMs available, developers need guidance in selecting the best tool for their specific workflows. This study provides a transparent, reproducible framework to evaluate which LLM is more suitable for particular use cases.

This methodology comprehensively addresses various dimensions of code quality assessment and evaluation. It provides a framework for systematically testing and comparing the code generation capabilities of large language models, and this approach can be utilized in future studies.

Future studies could apply this evaluation method to other programming languages and to other LLMs, aiding developers in selecting the best tool for code generation. As large language models (LLMs) continue to evolve, evaluating their code generation capabilities at a specific point in time can help guide better model selection decisions. Some other evaluation parameters like 'code comments present', 'code duplication exists' etc. can be added to result metrics. Factor like expanding the evaluation to include more programming languages would also make the assessment criteria more comprehensive.

REFERENCES

1. Ságodi Z, Siket I, Ferenc R. Methodology for code synthesis evaluation of LLMs presented by a case study of ChatGPT and copilot. IEEE Access. 2024 May 21; 12: 72303–16.
2. Helmuth T, Kelly P. PSB2: the second program synthesis benchmark suite. In Proceedings of the Genetic and Evolutionary Computation Conference. 2021 Jun 26; 785–794.
3. Bubeck S, Chandrasekaran V, Eldan R, Gehrke J, Horvitz E, Kamar E, Lee P, Lee YT, Li Y, Lundberg S, Nori H. Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv preprint arXiv:2303.12712. 2023 Mar 22.
4. Dieumegard A, Toom A, Pantel M. Model-based formal specification of a DSL library for a qualified code generator. In Proceedings of the 12th Workshop on OCL and Textual Modelling. 2012 Sep 30; 61–62.
5. Black P. Static analyzers: Seat belts for your code. IEEE Secur Priv. 2012 Jan 10; 10(3): 48–52.
6. Kaner C. Software engineering metrics: What do they measure and how do we know? In Proc Int'l Software Metrics Symposium, Chicago, IL, USA. 2004 Sep; 1–12.
7. Vaithilingam P, Zhang T, Glassman EL. 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. 2022 Apr 27; 1–7.
8. Al Madi N. How readable is model-generated code? examining readability and visual inspection of github copilot. In Proceedings of the 37th IEEE/ACM international conference on automated software engineering. 2022 Oct 10; 1–5.
9. Marcilio D, Furia CA, Bonifácio R, Pinto G. Automatically generating fix suggestions in response to static code analysis warnings. In 2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). 2019 Sep 30; 34–44.
10. Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). 2019 May 25; 209–219.
11. Zhang Y, Xiao Y, Kabir MM, Yao D, Meng N. Example-based vulnerability detection and repair in java code. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. 2022 May 16; 190–201.
12. Maddison C, Tarlow D. Structured generative models of natural source code. In International Conference on Machine Learning, PMLR. 2014 Jun 18; 649–657.

13. Sobania D, Briesch M, Rothlauf F. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In Proceedings of the genetic and evolutionary computation conference. 2022 Jul 8; 1019–1027.
14. Jain N, Vaidyanath S, Iyer A, Natarajan N, Parthasarathy S, Rajamani S, Sharma R. Jigsaw: Large language models meet program synthesis. In Proceedings of the 44th International Conference on Software Engineering. 2022 May 21; 1219–1231.
15. Pearce H, Tan B, Ahmad B, Karri R, Dolan-Gavitt B. Examining zero-shot vulnerability repair with large language models. In 2023 IEEE Symposium on Security and Privacy (SP). 2023 May 21; 2339–2356.
16. Asare O, Nagappan M, Asokan N. Is github's copilot as bad as humans at introducing vulnerabilities in code? Empir Software Eng. 2023 Nov; 28(6): 129.
17. White J, Hays S, Fu Q, Spencer-Smith J, Schmidt DC. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In Generative AI for Effective Software Development. Cham: Springer Nature Switzerland; 2024 Jun 1; 71–108.

