A Review on Code Generation with LLMs: Application and Evaluation

Jianxun Wang
National Engineering Research Center of Trustworthy Embedded Software
East China Normal University
Shanghai, China
jianxunwang@stu.ecnu.edu.cn

Yixiang Chen
National Engineering Research Center of Trustworthy Embedded Software
East China Normal University
Shanghai, China
yxchen@sei.ecnu.edu.cn

2023 IEEE International Conference on Medical Artificial Intelligence (MedAI) | 979-8-3503-5878-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/MEDAI59581.2023.00044

***Abstract*—Code generation is a longstanding subject in the field of computer science and software engineering, which aims at realizing an agent capable of writing code automatically aligning with human desire. With the booming development of large language models (LLMs) in recent years, code generation techniques powered by LLMs with strong coding ability have caught many researchers' interest. In this study, we conduct a review of recent studies about code generation with LLMs, from the application of LLM-based code generation to the evaluation of LLM-generated code. We find, with the powerful code understanding and writing ability LLMs provide, these novel techniques can be applied to manage various software engineering tasks, and indeed boost the productivity of developers to a great extent. But we also find, as an equally important subject, the evaluation receives less attention from researchers than the application. We conclude some limitations in existing studies about the evaluation of code generated by LLMs, like inadequate quality characteristics considered. And we think more effort is needed to narrow the gap between research on the evaluation and the application.**

***Index Terms*—large language models (LLMs), code generation, code completion, automatic program repair, code quality evaluation**

I. Introduction

Code generation, also called program synthesis sometimes, is a fascinating topic for researchers in computer science and software engineering. A machine capable of writing desired code automatically according to human demand is the researchers' ultimate purpose. With consistent efforts for decades, they have made great progress, but there is still a great distance from fully automatic and practical code generation, especially generating code based on natural language descriptions. However, the emergence of large language models (LLMs) in recent years, the GPT series in particular, provides them with a new possibility to achieve this ultimate purpose.

With a massive number of parameters up to 175B, terabyte-level training data, and well-designed architecture, LLMs were endowed with unprecedentedly powerful language understanding, processing, and generating capability. Coincidentally, in order to make code more understandable for humans, most program languages, especially high-level program languages, are designed based on natural languages. Therefore, it is intuitive to think LLMs trained on massive existing code can become able to read and write code smoothly, and this exciting assumption has been proven true by researchers in recent years. With sufficient code and descriptions from human programmers in training data, LLMs can generate code according to given natural language descriptions about target code, incomplete code segments to be filled or even buggy code snippets to be fixed. And after obtaining complete code thoroughly generated by LLMs, to what extent is this code trustworthy becomes a new considerable question for researchers.

In this study, we mainly review 20 existing studies about code generation with LLMs in the past five years. We review these studies from two aspects: the application of code generation with LLMs and the evaluation of code generated by LLMs. In terms of application, we focus on three prevailing kinds: description to code, code completion, and automatic program repair. As for evaluation, we discuss related studies based on the quality characteristics of generated code each study concerns, which are accordingly divided into three parts: functional correctness, security, and others (some quality characteristics considered less by these studies). These studies we review are presented in Table I, categorized by the subsection each is discussed in this paper.

TABLE I
Overview of Studies Reviewed by Us

Section[a]	Subsection[b]	Reviewed Studies
Application	*Description to Code*	[12] [13] [14]
	Code Completion	[15] [16] [17]
	Automatic Program Repair	[19] [20] [21]
Evaluation	*Functional Correctness*	[5] [25] [27] [28] [29]
	Security	[30] [31] [32] [33]
	Others	[34] [36]

[a]Each refers to the corresponding section in this paper, where Application and Evaluation indicate section III and IV respectively.
[b]Each refers to the corresponding subsection in this paper.

By reviewing these studies, we find there is a significant gap between research on the application and the evaluation of LLM-based code generation. Many pioneer researchers have been exploiting the potential of code generation with LLMs in actual applications, and have gained lots of amazing results.

979-8-3503-5878-0/23/$31.00 ©2023 IEEE
DOI 10.1109/MedAI59581.2023.00044

However, as a subject of equal importance, the evaluation of LLM-generated code is less noticed by researchers, and the current state of related research can hardly catch up with the former. We finally find existing research on the evaluation shares some limitations, which are critical to be addressed in order to close the gap soon.

II. Background and Motivation

A. Large Language Models (LLMs)

In the field of natural language processing (NLP), language models are usually designed for some specific tasks, like machine translation, information extraction, question answering etc [1]. However, large language models (LLMs), which are machine learning-based language models with a large number of parameters, and trained on massive data, not only have a more powerful capability but also can handle multiple kinds of tasks without retraining.

To illustrate LLMs further, we have to mention the GPT series [2] [3] [4] [7] [8], which contains several most prevailing language models in recent years. When given a string of tokens within its context window, GPT-3 [4] can predict the most possible token next according to its abundant training corpus. Thus, users can directly communicate with GPT-3, or use task descriptions as prompts to ask GPT-3 to do specific downstream tasks. Later, in order to boost the code writing ability, OpenAI retrained GPT-3 on massive open source codes from GitHub [5]. The new language model they got was named Codex, which can complete code based on existing code or generate code snippets according to natural language descriptions. After that, GitHub released Copilot [6], an AI pair programmer powered by Codex, which can draw context from comments and code to suggest individual lines or whole functions instantly. In the following GPT-3.5 (ChatGPT) [7] and GPT-4 [8], the amazing ability of Codex was integrated and enhanced.

B. Code Generation

Code generation, also known as program synthesis, meaning the automatic construction of software or self-writing code [9], is a longstanding topic discussed in the fields of software engineering, computer science, programming language, and artificial intelligence. Code generation aims at automatically generating code according to given descriptions or existing code.

For a long time in the past, most research about code generation was limited to domain-specific languages (DSLs) and programmers had to write tedious formal specifications as descriptions for generation [10]. However, with the development of artificial intelligence, lots of machine learning and deep learning techniques are applied to code generation, making it more powerful and flexible [11]. Since most programming languages, especially high-level programming languages, share many similarities with natural language, it is intuitive to utilize powerful language models in code generation, and this idea has been proven very effective by studies in recent years.

C. Motivation

This study is motivated by our practical experience of using LLM-based code generation tools, especially Copilot developed by GitHub. We are surprised by the powerful code generation ability of Copilot, which can predicate and complete our code smoothly, and support a large range of programming languages. This experience drives us to explore more amazing applications of this new technique.

In order to know more about it, we asked Copilot (*VSCode extension: GitHub Copilot v1.105.350*) to complete a heap-sort C++ function (the function signature as prompt) and tested it with an unordered array `[9,1,12,0,-5,25,17,-2,6,21]`. We found Copilot generated the wrong code and its output result is wrong, as shown in Fig. 1. Moreover, the generated code snippet does not contain any comments, which makes it difficult for programmers to check and modify it manually. This worrying outcome inspires our concerns about the evaluation of LLM-generated code.

```
void heap_sort(int array[], int left, int right) {
    int i = left, j = 2 * i + 1;
    int temp = array[i];
    while (j <= right) {
        if (j < right && array[j] < array[j + 1]) {
            j++;
        }
        if (temp < array[j]) {
            array[i] = array[j];
            i = j;
            j = 2 * i + 1;
        } else {
            break;
        }
    }
    array[i] = temp;
}
```

Test Output: 12 1 25 0 -5 9 17 -2 6 21 ✗

Fig. 1. The Heap-Sort C++ Function Completed by Copilot and The Test Output.

III. Application of Code Generation with LLMs

With surprising natural language processing ability, LLMs can not only read and write code but also handle documentation, comments, warnings, or error messages generated along with the development of software. Thus, code generation with LLMs has great potential in automatically managing various software engineering tasks. In recent years, researchers have been exploiting its ability and have found many amazing applications of it. We discuss three prevailing applications here: **Description to Code**, **Code Completion**, and **Automatic Program Repair**, as shown in Fig. 2. Besides these three, there are some other applications also attractive, like code translation, test generation, documentation generation, etc.

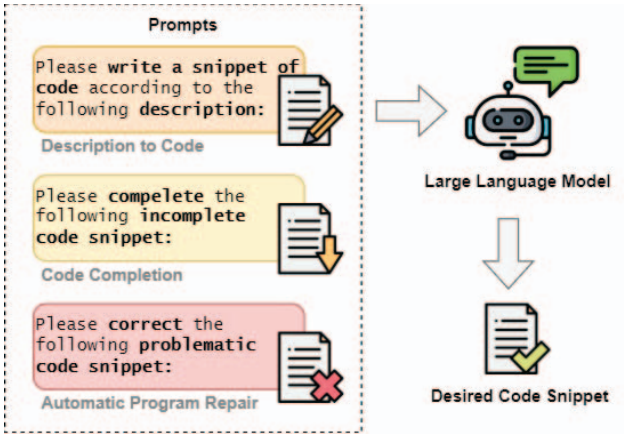

Fig. 2. Three Prevailing Applications: **Description to Code**, **Code Completion**, and **Automatic Program Repair**

A. Description to Code

Nothing will be more exciting for software developers than a machine that can generate code according to given descriptions written in natural language. Some researchers have been trying this with several state-of-the-art LLMs.

Finnie-Ansley et al. [12] applied Codex mentioned above to introductory programming. They used 23 questions from two introductory programming course tests as descriptions for Codex to generate Python code. They found Codex performed better than most student participants in these two tests and its solutions include lots of variations. They thought LLMs with the ability to write code like Codex could be an opportunity and also a threat to introductory programming education.

Jiang et al. [13] developed a natural language code synthesis tool GetLine, and conducted a user study with it. GetLine is backed by LaMDA and provides a user interface. Users can input natural language requests and select a target programming language, and then GetLine can generate multiple outputs for users to choose from. Finally, the authors concluded several useful implications of future code synthesis tool design from their user study.

Dong et al. [14] proposed a self-collaboration framework for LLMs to enhance their capability of solving coding problems. They asked three ChatGPT instances to play analyst, coder, and tester along the development process of software respectively, and coding problems were fed as user requirements. Then, these three roles can interact and collaborate by chatting to generate code. By conducting a comprehensive method, they found the performance of this self-collaboration code generation is 30% higher than the naive direct approach.

B. Code Completion

Code completion is an indispensable feature for integrated development environments (IDEs), which can offer developers code suggestions according to available contextual information. The ability of code completion has been staying at the syntactic level for a long time, and the promotion of productivity it brings is limited. However, LLMs, designed for NLP, can naturally predict what users might type next based on existing code and comments just like word prediction, lifting the ability of code completion to the semantic level. And Copilot, the powerful code completion tool based on Codex, attracts many researchers' and developers' attention.

Moroz et al. [15] studied the applications and shortcomings of Copilot. They found Copilot has many advantages as a programming assistant, such as Copilot can be a good assistant for skilled programmers, as well as novice developers. They also found some problems and gaps in its applications like unchecked low-quality code. Overall, they suggested Copilot still has lots of growth opportunities, more effort should be taken to make it safer, more reliable, and more effective.

Ziegler et al. [16] conducted a case study with Copilot to find out its impact on the productivity of users. Combining these objective usage data and the subjective perceptions of developers, they found suggestions' acceptance rate can be a great predictor of productivity developers perceive, which can reflect the perception of users to some extent. Moreover, they found acceptance rate varies among developers, depending on their behavior.

Barke et al. [17] conducted a grounded theory analysis, aiming at knowing how programmers interact with code-generative models, like Copilot. They found users' interactions with Copilot can be classified into two modes—acceleration and exploration. Acceleration mode can boost users' productivity greatly and exploration mode can always help users handle unfamiliar tasks. Based on this finding, they also proposed some recommendations for users.

C. Automatic Program Repair

Because of the high cost of traditional software maintenance approaches, which occupies over half of the software development lifecycle [18], automatic program repair (APR) techniques have been studied by many commercial companies and academic institutions for years. The booming development of LLMs in recent years undoubtedly provides a new possibility for APR.

According to Xia and Zhang [19], template-based tools have the best performance among traditional APR methods, which fix bugs by matching specific buggy code patterns and applying corresponding patches. But in this way, the capability of template-based tools will be limited by the coverage of their finite pattern base. To overcome this issue, some researchers apply machine learning techniques to APR, recognizing bug fixing as a neural machine translation task, which translates buggy code into correct code. Thus, the emerging LLMs capable of handling various NLP tasks can be a promising solution.

Kolak et al. [20] noticed the potential capability of language models in distinguishing bugs and patches, so they studied the performance of LLMs with different scales in program repair tasks. They selected three publicly available versions of PolyCoder and Codex. Then they tested these four models on 80 buggy programs. According to their result, they found

larger models are more successful patch generators and tend to generate patches similar to a real developer.

Sobania et al. [21] evaluated and analyzed the performance of ChatGPT (GPT-3.5) in bug fixing. They selected all Python problems from QuixBugs [22] benchmark, asking ChatGPT for the bug fix four times for each problem. Compared with previous approaches, ChatGPT notably outperformed traditional APR tools and is competitive with other LLM-based approaches, like Codex. Their result also shows that, as a dialogue system, with further information like error messages, ChatGPT can outperform the state-of-the-art.

Xia and Zhang [19] proposed a fully automated conversation-driven APR approach—CHATREPAIR, based on ChatGPT. This approach keeps asking ChatGPT to generate patches and giving detailed test results back until the correct patch is obtained. By evaluating CHATREPAIR against other APR tools on two widely studied benchmarks, they surprisedly found, though as a conversational APR approach, CHATREPAIR obtained the state-of-the-art performance.

IV. EVALUATION OF CODE GENERATED BY LLMS

Software source code quality has been an important subject in the field of software engineering for decades, which can be traced back to the 1960s [23]. Many researchers have proposed various metrics, methods, and models for source code quality evaluation in different situations, which is also called software quality evaluation at the source level. These evaluation approaches have become the basis of software source code quality assurance and promotion practices in the industry today.

When researchers noticed the great potential of LLMs in writing code, concerns about the quality of LLM-generated code began to emerge at the same time. If unreliable code generated by LLMs is directly introduced into the software without carefully checking, this may lead to some disastrous outcomes like system breakdown or privacy data leak. Therefore, there has been much research in recent years focusing on evaluating LLM-generated code from the perspective of software source code quality.

Software quality consists of multiple quality characteristics (sometimes also called quality factors or quality attributions), such as functionality suitability, performance efficiency, compatibility, etc. according to ISO/IEC 25010 [24], and lots of corresponding sub-characteristics. Thus we review these researches about LLM-generated source code evaluation and present them according to the quality characteristics or sub-characteristics each study concerns. Specifically, we present them in three categories: **Functional Correctness**, **Security**, and **Others**.

A. Functional Correctness

As mentioned above, automatic code generation is a long-standing topic that has been studied for decades. In order to access the performance of a newly proposed code generation approach and compare it with other existing ones, a proper code evaluation metric (CEM) is indispensable for the development of code generation techniques, and there are mainly two types of prevailing CEMs: match-based and execution-based [25].

Since code generation can be viewed as the translation from description to code, the match-based evaluation metrics commonly used in machine translation can be helpful, which evaluate by calculating the similarity between generated sentences and reference sentences, such as **BLEU** [26] and **CodeBLEU** [27]. However, these match-based CEMs have deficiencies in reflecting the functional correctness of generated code, because these CEMs are unable to decide the functional equivalence between generated code and reference code [25]. Therefore, more execution-based CEMs were proposed by later researchers.

Currently, there are two most commonly used execution-based CEMs to evaluate the functional correctness of code: **AvgPassRatio** [28] and **Pass@k** [29], which both depend on the execution of generated code on a prepared test set. In order to evaluate Codex they developed, Chen et al. [5] released a new evaluation problem set HumanEval and calculated the Pass@k value.

Because of the need for repeated executions on test cases, Dong et al. [25] thought the computation of execution-based CEMs is costly, slow, and insecure, while match-based CEMs are inaccurate, so they proposed a new LLM-based CEM—CodeScore, for functional correctness evaluation. They defined PassRatio according to AvgPassRatio and made CodeScore an execution-free alternative for PassRatio. By evaluating with their LLM-based framework, CodeScore outperforms match-based CEMs in accuracy compared with PassRatio and costs much less time than execution-based CEMs.

B. Security

The powerful code generation capability of LLMs relies on massive available code snippets in training data, which are mostly from open-source code repositories. However, it is likely there exist potential vulnerabilities or even malicious code snippets in these open-source codes, which may leak into the output of LLMs and harm the security of developed software. Many researchers have similar worries and evaluate the security of LLM-generated code from various aspects.

Pearce et al. [30] focused on the security of Copilot's code contributions. First, based on MITRE's Common Weakness Enumerations (CWEs), they created a prompt dataset containing various security-relevant scenarios. The security of the completed code was evaluated with the automated analysis tool and manual inspection. They found that, in about 44% of all scenarios, Copilot did generate code with a relevant weakness, and some of the weaknesses are introduced more frequently.

Sandovalet et al. [31] conducted a user study to find out whether student programmers will write more insecure code with the help of an LLM-based code assistant. Considering real-world programming is mostly project-based, they design a "shopping list" C program completion task. They found the

number of server security bugs produced by LLM-assisted participants is no greater than 10% than the control.

Asare et al. [32] conducted a comparative empirical analysis to find out whether Copilot is as bad as human developers in introducing vulnerabilities when writing code. Based on a dataset of C/C++ vulnerabilities from several projects in the real world, they recreated the same scenario for Copilot by deleting bug/patch-relevant code. They found Copilot introduced the same vulnerabilities as humans only one-third of the time, which is not as bad as human developers.

Khoury et al. [33] experimented to evaluate the security of code generated by ChatGPT. They designed 21 problems across 5 programming languages, and each problem is prone to introduce a specific vulnerability in CWEs when solving. In conclusion, they found ChatGPT can frequently generate insecure code and experienced programmers are still irreplaceable to produce code reliable enough.

C. Others

There are also some researchers concerning other aspects of source qualities besides functional correctness and security.

Understandability. Understandablity reflects how easily programmers can fully understand the logic and function of a specific code snippet. Nguyen et al. [34] conducted an empirical study to evaluate the correctness and understandability of code generated by Copilot. In terms of the evaluation of understandability, they calculated the cyclomatic complexity and cognitive complexity of generated code, which both positively correlate with understandability according to previous study [35]. Overall, they thought Copilot could produce easily understandable code under this experimental circumstance, but their data may be not enough to get a general conclusion.

Maintainability and Readability. As the definition given by ISO/IEC 25010 [24], maintainability represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it, or adapt it to changes in environment and requirements. Readability, which is similar to understandability, evaluates the complexity of code in the syntactic aspect, while understandability conducts evaluations in the dynamic aspect [35]. Siddiq et al. [36] conducted an empirical study on code smells in training code and generating code of LLM-based code generation techniques, as well as the relation between them. Code smells can include security issues, design decision issues, and coding standard violations in source code, which are patterns that indicate lower maintainability and readability. According to the results, they thought bad code patterns in training code do leak to the code generated by models because they found the type of code smells in generating code is a subset of those in training code.

V. Discussion

With the rapid development of code generation with LLMs, many researchers have been exploring its possibility in practice. As we have reviewed in section III, LLM-based code generation techniques have astonishing potential and various applications in managing software engineering tasks, which can greatly boost the productivity of developers. However, as an equally critical subject, the evaluation of LLM-generated code fails to keep up with the application. By reviewing related research in section IV, we found the following limitations in existing studies about the evaluation of code generated by LLMs.

Inadequate quality characteristics for evaluation. According to ISO/IEC 25010 [24], there are plenty of quality characteristics and sub-characteristics to evaluate code from different aspects. However, most existing research only focuses on the functional correctness and security issues of LLM-generated code. We think that's because these two are truly the most elementary and concerning aspects of code, also the most easily perceived by programmers and users of final software products. However, many other characteristics can also impact the integral quality of code to a great extent, like compatibility, maintainability, portability, etc.

Lack of systematic and quantitative evaluation model. As for traditional code, there are many studies proposing systematic quality [37] (or similarly trustworthiness [38]) evaluation models, which can produce a quantitative evaluation result about the quality of code. However, to the best of our knowledge, there is still no research imposing these traditional methods on the evaluation of LLM-generated code. Though there may be some challenges to be conquered, we believe these traditional models can bring many benefits to current research on LLM-generated code evaluation.

Ignoring human engagement when conducting the evaluation. According to Sandovalet et al. [31], there are generally two kinds of software development modes with LLMs: the autopilot mode and the assisted mode. Due to limited functional correctness and potential vulnerabilities of LLM-generated code, the assisted mode is a more practical way at the present phase, like the practices of Copilot and ChatGPT. As for the evaluation of code generated in this way, human engagement should be taken into consideration as well, because their prompt strategies, suggestions selections, and some other behaviors do affect the final code. However, from our review, we found most research failed to consider the role of human programmers when conducting the evaluation.

Lack of specific research on evaluation. We found that in most existing research mentioning LLM-generated code quality evaluation, the initial motivation of evaluation is to measure and compare the performance of LLMs in terms of code generation. For instance, functional correctness, which is most used by researchers, has been a golden metric to embody the code-writing ability of LLMs. Therefore, there is a lack of specific research focusing on the evaluation of code generation by LLMs, and we think they are much needed because specific research on evaluation is meaningful not only for developers of LLMs but also for users, which can help programmers evaluate the trustworthiness of generated code.

In our future work, we will recognize LLM-assisted coding as a brand-new software development approach and evaluate the trustworthiness of the resulting software. With this new development process, we have to make some necessary changes

to the traditional software trustworthiness model to adapt to it. We will focus on not only the generated source code but also the whole process assisted with LLMs, including the capability of a specific LLM, the interaction between human developers and models and so on.

VI. Conclusion

In this study, we review recent research on code evaluation with LLMs from two aspects: the application of code generation with LLMs and the evaluation of code generated by LLMs. We find, with the help of recent emerging powerful LLMs, code generation techniques can successfully handle more complex tasks than before, and many researchers proposed various novel applications of LLM-based code generation. However, research around the evaluation of LLM-generated code fails to keep pace with the application. By reviewing a limited amount of related studies, we found some limitations in the current research stage, and we think more effort is needed to fill the gap in the future.

References

[1] S. J. Russell et al., Artificial intelligence: a modern approach, Fourth edition, Global edition. in Pearson series in artificial intelligence. Harlow: Pearson, 2022.

[2] A. Radford and K. Narasimhan, “Improving Language Understanding by Generative Pre-Training,” 2018.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” 2019.

[4] T. B. Brown et al., “Language Models are Few-Shot Learners.” arXiv, Jul. 22, 2020. doi: 10.48550/arXiv.2005.14165.

[5] M. Chen et al., “Evaluating Large Language Models Trained on Code.” arXiv, Jul. 14, 2021. doi: 10.48550/arXiv.2107.03374.

[6] "GitHub Copilot · Your AI pair programmer," GitHub. https://github.com/features/copilot (accessed Jun. 20, 2023).

[7] "ChatGPT." https://openai.com/chatgpt (accessed Jun. 20, 2023).

[8] "GPT-4." https://openai.com/gpt-4 (accessed Jun. 20, 2023).

[9] C. David and D. Kroening, “Program synthesis: challenges and opportunities,” Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, vol. 375, no. 2104, p. 20150403, Sep. 2017, doi: 10.1098/rsta.2015.0403.

[10] Gu B, Yu B, Dong XG, Li XF, Zhong RM, Yang MF. Intelligent program synthesis techniques: Literature review. Ruan Jian Xue Bao/Journal of Software, 2021,32(5):1373-1384 (in Chinese). http://www.jos.org.cn/1000-9825/6200.htm

[11] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," IEEE Access, vol. 10, pp. 82434-82455, 2022, doi: 10.1109/ACCESS.2022.3196347.

[12] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming,” in Proceedings of the 24th Australasian Computing Education Conference, in ACE ’22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 10-19. doi: 10.1145/3511861.3511863.

[13] E. Jiang et al., “Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models,” in Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, in CHI ’22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1-19. doi: 10.1145/3491102.3501870.

[14] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration Code Generation via ChatGPT.” arXiv, Apr. 15, 2023. Accessed: Apr. 21, 2023. [Online]. Available: http://arxiv.org/abs/2304.07590

[15] E. A. Moroz, V. O. Grizkevich, and I. M. Novozhilov, “The Potential of Artificial Intelligence as a Method of Software Developer’s Productivity Improvement,” in 2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), Jan. 2022, pp. 386-390. doi: 10.1109/ElConRus54750.2022.9755659.

[16] A. Ziegler et al., “Productivity Assessment of Neural Code Completion.” arXiv, May 13, 2022. doi: 10.48550/arXiv.2205.06537.

[17] S. Barke, M. B. James, and N. Polikarpova, “Grounded Copilot: How Programmers Interact with Code-Generating Models.” arXiv, Oct. 31, 2022. doi: 10.48550/arXiv.2206.15000.

[18] Jiang JJ, Chen JJ, Xiong YF. Survey of automatic program repair techniques. Ruan Jian Xue Bao/Journal of Software, 2021,32(9):2665-2690 (in Chinese). http://www.jos.org.cn/1000-9825/6274.htm

[19] Chunqiu Steven Xia and Lingming Zhang, “Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT.” arXiv, Apr. 01, 2023. doi: 10.48550/arXiv.2304.00385.

[20] S. Kolak, R. Martins, C. L. Goues, and V. J. Hellendoorn, “PATCH GENERATION WITH LANGUAGE MODELS: FEASIBILITY AND SCALING BEHAVIOR,” 2022.

[21] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An Analysis of the Automatic Bug Fixing Performance of ChatGPT.” arXiv, Jan. 20, 2023. doi: 10.48550/arXiv.2301.08653.

[22] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge,” in Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity, 2017, pp. 55-56.

[23] Yikang Shao, Wu Liu, Jun Ai, and Chunhui Yang, “A Quantitative Measurement Method of Code Quality Evaluation Indicators based on Data Mining,” in 2022 9th International Conference on Dependable Systems and Their Applications (DSA), Aug. 2022, pp. 659-669. doi: 10.1109/DSA56465.2022.00094.

[24] “ISO 25010.” https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (accessed Jun. 24, 2023).

[25] Y. Dong, J. Ding, X. Jiang, Z. Li, G. Li, and Z. Jin, “CodeScore: Evaluating Code Generation by Learning Code Execution.” arXiv, Jan. 21, 2023. doi: 10.48550/arXiv.2301.09043.

[26] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: a method for automatic evaluation of machine translation,” in Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, in ACL ’02. USA: Association for Computational Linguistics, Jul. 2002, pp. 311-318. doi: 10.3115/1073083.1073135.

[27] S. Ren et al., “CodeBLEU: a Method for Automatic Evaluation of Code Synthesis.” arXiv, Sep. 27, 2020. doi: 10.48550/arXiv.2009.10297.

[28] D. Hendrycks et al., “Measuring Coding Challenge Competence With APPS.” arXiv, Nov. 08, 2021. doi: 10.48550/arXiv.2105.09938.

[29] S. Kulal et al., “SPoC: Search-based Pseudocode to Code.” arXiv, Jun. 11, 2019. doi: 10.48550/arXiv.1906.04908.

[30] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions,” in 2022 IEEE Symposium on Security and Privacy (SP), May 2022, pp. 754-768. doi: 10.1109/SP46214.2022.9833571.

[31] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, “Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants,” Aug. 2022.

[32] O. Asare, M. Nagappan, and N. Asokan, “Is GitHub’s Copilot as Bad as Humans at Introducing Vulnerabilities in Code?” arXiv, Feb. 14, 2023. doi: 10.48550/arXiv.2204.04741.

[33] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, “How Secure is Code Generated by ChatGPT?” arXiv, Apr. 19, 2023. doi: 10.48550/arXiv.2304.09655.

[34] N. Nguyen and S. Nadi, “An empirical evaluation of GitHub copilot’s code suggestions,” in Proceedings of the 19th International Conference on Mining Software Repositories, in MSR ’22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 1-5. doi: 10.1145/3524842.3528470.

[35] C. E. C. Dantas and M. A. Maia, “Readability and Understandability Scores for Snippet Assessment: an Exploratory Study.” Aug. 20, 2021. doi: 10.5281/zenodo.5224346.

[36] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, “An Empirical Study of Code Smells in Transformer-based Code Generation Techniques,” in 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Oct. 2022, pp. 71-82. doi: 10.1109/SCAM55253.2022.00014.

[37] Meng Yan, Xin Xia, Xiaohong Zhang, Ling Xu, Dan Yang, and Shanping Li, “Software quality assessment model: a systematic mapping study,” Sci. China Inf. Sci., vol. 62, no. 9, p. 191101, Jul. 2019, doi: 10.1007/s11432-018-9608-3.

[38] Yixiang Chen, Hongwei Tao. Software Trustworthiness Measurement Evaluation and Enhancement Specification. Beijing: Science Press, 2019. (in Chinese).