ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

AI-Powered Code Generation Evaluating the Effectiveness of Large Language Models (LLMs) in Automated Software Development

Ravikanth Konda

Senior Software Developer, USA

ABSTRACT

The rapid evolution of Artificial Intelligence (AI) has brought about significant advancements in multiple domains, including software development. One of the most promising innovations is AI-powered code generation through Large Language Models (LLMs), such as OpenAI's GPT-3 and GPT-4. These models, having been trained on large amounts of programming data, have the ability to produce human-readable code from natural language inputs, which is a big potential for simplifying and optimizing software development processes. The aim of this paper is to analyze the performance of LLMs in automated software development by testing their performance on a variety of tasks such as code generation, debugging, and optimization of software. The research explores both the strengths and weaknesses that these models have to offer, in terms of some of the most important indicators like code quality, generation time, and maintainability of the code. According to our observation, although LLMs hold immense potential to automate mundane programming tasks and enhance developer productivity, they still struggle to cope with more intricate, domain-specific programming tasks involving a higher level of understanding, for example, designing architectures and top-level decision-making. In spite of such shortcomings, LLMs can tremendously enhance software development processes, particularly for small-scale projects or act as helpers for more senior developers. The paper summarizes by reflecting on the potential for LLMs to transform software development processes in the future, while also the importance of the model's reliability, coding quality, and security to be improved if it is to be made applicable to larger, more crucial uses.

***Corresponding author**

Ravikanth Konda, Senior Software Developer, USA.

Received: June 05, 2023; **Accepted:** June 09, 2023; **Published:** June 16, 2023

Keywords: AI-Powered Code Generation, Large Language Models (LLMs), Automated Software Development, Machine Learning, Software Development Tools, Code Synthesis, Deep Learning, GPT Models, Programming Assistance, Software Engineering, Automation in Software Engineering, AI-Driven Development, Software Optimization

Introduction

The software development industry has undergone explosive growth in the last few decades due to the accelerating growth of computing power, the emergence of cloud-based platforms, and the advancements in programming languages and development frameworks. In the last few years, one of the most revolutionary drivers in this space has been the adoption of Artificial Intelligence (AI) in different parts of the software development process. Some of the most revolutionary developments include the introduction of Large Language Models (LLMs), including GPT-3 and GPT-4, which can produce executable, human-understandable code based on natural language specifications. These models show tremendous progress toward achieving automated software creation.

Typically, coding has demanded that human programmers need to have profound knowledge of programming languages, algorithms, data structures, and design principles of software. Even mundane development activities like boilerplate code writing, building APIs, or debugging take time, focus, and skill. The advent of AI-based tools has the potential to automate most of these repetitive or mechanical activities, freeing developers to work on more creative and strategic aspects of software development. LLMs, through the utilization of enormous collections of code and natural language, are now capable of understanding high-level requirements and producing syntactically correct and functionally relevant code with very little human intervention.

The range of applications of LLMs in software development is enormous. From creating frontend elements in HTML/CSS/JavaScript, to creating backend code in languages such as Python, Java, or C#, LLMs are being utilized in the entire software stack. Their potential to aid code completion, refactoring, debugging, test generation, and even code documentation has generated a burgeoning interest in industry and academia alike. GitHub Copilot, for example, already exists as a tool grounded in OpenAI Codex (a fine-tuned version of GPT), which has already been inserted into actual development environments to generate AI-powered suggestions and automate large parts of the coding process.

But with this new power comes also a set of subtle problems and questions. Can these models be relied upon to produce secure, optimized, and maintainable code? Do they comprehend the meaning behind a developer's instructions thoroughly enough to produce suitable solutions? What are the ethics of depending upon AI in the task of programming, particularly in systems of high importance such as healthcare, finance, or aviation? Also, there is the issue of AI-generated code intellectual property, potential learned bias or vulnerabilities, and the possibility of job displacement in the software development sector.

The purpose of this research paper is to critically analyze the capability of LLMs in automatic software development. It targets three main goals: (1) evaluating the strengths of LLMs in producing correct, readable, and maintainable code for a wide range of problem domains; (2) analyzing the shortcomings and difficulties of using LLMs, especially concerning scalability, context understanding, and quality control; and (3) investigating the consequences of mass adoption of LLMs on the software development process, industry standards, and programming's future.

To attain these objectives, the research employs a mixed-methods design, incorporating empirical examination of code produced by AI with developer opinions obtained through surveys and interviews. By comparing the performance of cutting-edge models such as GPT-3, Codex, and GPT-4 on tasks of different complexity levels, and against code produced by human developers, the paper presents a balanced assessment of LLM effectiveness.

In so doing, this research adds to an increasing body of evidence on AI-facilitated software engineering and aims to guide future research, policy, and best practice surrounding the use of LLMs in software development. As such technologies develop further, knowledge of their strengths and weaknesses will be critical to using their potential responsibly and effectively.

Literature Review

The advent of Large Language Models (LLMs) represents a paradigm shift in software development. These models, which are trained on massive corpora of code and natural language, have the potential to automate various phases of the development cycle—ranging from generating boilerplate code to producing intricate logic. Recent breakthroughs in transformer-based architectures have further improved their capacity to produce syntactically and semantically correct code. This section discusses the evolution of code generation with AI, including cornerstone models, testing methodologies, operational challenges, and academic and industry research contributions.

Code Generation Foundations in LLMs

Vaswani' et al. introduction of transformer architectures made LLMs possible [1]. The self-attention mechanism proposed in their groundbreaking paper "Attention is All You Need" transformed natural language understanding and generation, facilitating the creation of high-capacity models such as GPT, BERT, and Codex. Brown et al. expanded on this concept by showing that large transformers, including GPT-3, were capable of few-shot learning with few examples [2]. These early studies provided the theoretical foundation for utilizing such models for code generation.

The idea of converting natural language to code (NL2Code) developed with research such as that by Chaudhary et al., who gave a detailed overview of early AI tools for code generation. Their research outlined the development of symbolic AI techniques to deep learning-based techniques and presented tools like DeepCoder, AutoML, and Neural Code Summarizer.

Recent Model Advances and Tools

More attention has been directed in recent years toward specialist models like OpenAI Codex, AlphaCode from Google, CodeGen by Salesforce, and Code LLaMA by Meta. They learn from billions of lines of code on bases like GitHub and Stack Overflow. Zan et al. offered a comparative evaluation of 27 of these models, highlighting the need for dataset diversity, size, and fine-tuning methods [3]. Their results indicate that larger models trained on domain-specific data perform substantially better in benchmark tests.

GitHub Copilot, based on Codex, is likely the most popular AI-powered coding assistant. It plugs directly into IDEs such as Visual Studio Code and supports a variety of languages, including Python, JavaScript, and Go. Ribeiro et al. tested Copilot on actual programming problems and reported that it had high syntactic correctness and tended to produce innovative, functional solutions—although it still needed human supervision [4].

Evaluation Strategies and Benchmarks

Thorough evaluation is essential for determining LLM reliability. Zhang et al. presented an exhaustive investigation into code generation evaluation, with the authors recommending a hybrid approach of combining syntactic measures (e.g., BLEU, CodeBLEU) and semantic measures (e.g., execution accuracy, test case pass rate) [5]. Frameworks like HumanEval, MBPP (Mostly Basic Python Problems), and APPS (Automated Programming Progress Standard) are extensively used to evaluate LLM performance in various task types.

Jiang et al. emphasized execution-based evaluation [6]. BLEU scores might reflect structural similarity, yet only real execution demonstrates whether or not the synthesized code addresses the desired problem. Their work encourages dynamic evaluation platforms wherein code is exercised against functional test cases.

Yang et al. classified deep-learning-based code generators into retrieval-based, generative, and hybrid models [7]. Retrieval-augmented generators employ contextual search across code repositories to return relevant code snippets prior to code generation. Such hybrid methods that integrate semantic search with generation have exhibited improved generalization, particularly for domain-specific tasks.

Challenges and Limitations

Even though they hold promise, LLMs have some limitations. One of the significant issues is semantic inaccuracy—models can generate syntactically correct but logically incorrect code. Ribeiro et al. noted that LLMs tend to hallucinate non-existent functions or APIs, particularly when subjected to unfamiliar or ambiguous inputs [4]. Furthermore, the performance of LLMs becomes poor when subjected to multi-step reasoning or long-context problems.

Model bias is another challenge. As LLMs are trained on open-source repositories, they can mirror outdated or insecure practices. They can also perpetuate bias in variable names, comments, or even data structures. Chaudhary et al. highlight that this bias, while subtle, can build up and spread to production environments [8].

The intellectual property (IP) problem also hangs over. Since LLMs are trained on public code, they can reproduce copyrighted or licensed material verbatim. This has led to legal concerns, especially surrounding such tools as Copilot. Developers have to go through the laborious process of checking that generated code doesn't violate licensing terms, an activity that can undermine the productivity gains.

Ethical and Practical Considerations

The swift embrace of LLM-based coding assistants has created a combination of enthusiasm and anxiety. While these tools can increase developer productivity, they raise ethical issues regarding authorship, attribution, and accountability. If a model creates a bug or vulnerability, it's debatable whether responsibility is attributed to the developer, the model, or its developers.

Practically speaking, some organizations have started exploring hybrid development paradigms in which LLMs serve as junior developers or co-pilots. Such configurations have resulted in higher throughput for activities like writing unit tests, creating documentation, or migrating legacy code. Zhang et al. contend, however, that developers need to be trained not only in coding but also in prompt engineering to make optimal use of these tools [5].

Future Directions and Research

Future studies should close the loop between semantic understanding and syntactic fluency. This can be done through fine-tuning models with execution feedback, human-in-the-loop frameworks, or reinforcement learning. Jiang et al. advocate for creating LLMs that are modular and can be dynamically reconfigured to execute specific tasks like debugging, refactoring, or security auditing [6].

Additionally, industry and academia collaboration can further the creation of open benchmarks and clear datasets. This would enable more reproducible and fair research in AI-powered software development.

As LLMs become more advanced, there is also potential to investigate cross-domain uses—mashing up LLMs with DevOps, cyber-physical systems, and real-time collaborative development platforms. This has the potential to reframe the software engineering pipeline and create a new generation of intelligent software agents.

Methodology

To comprehensively test the performance of Large Language Models (LLMs) in software development automation, this research uses a comparative, experimental design combining quantitative measurement with qualitative observation. The main goal is to measure how accurately LLMs generate code based on correctness, efficiency, and usability, and investigate how model architecture, training data, and prompt design affect the quality of the output.

The research was conducted using a set of widely recognized and publicly available LLMs that are specifically designed for code generation or have demonstrated strong coding capabilities. These include OpenAI's Codex (a variant of GPT-3 fine-tuned on programming data), Salesforce's CodeT5, and Facebook's PLBART. Each of these models represents a different design philosophy—ranging from transformer-based encoder-decoder structures to autoregressive models—and allows for an insightful comparison across architectures. All the tests were run in a uniform computational setting with the same prompts and standards to avoid extrinsic variability and guarantee reproducibility.

Tasks were picked from well-known code testing benchmarks such as HumanEval, MBPP (Mostly Basic Python Problems), and CodeXGLUE. These benchmarks provide a broad variety of programming problems that examine algorithmic reasoning, data structure manipulation, and implementation correctness across languages like Python, JavaScript, and Java. Every task has a natural language prompt after which comes either an empty code stub or the signature of a function, inducing the model to produce correct and complete code. Crucially, these benchmarks come with unit tests or ground truth outputs, with which the accuracy of every produced response was confirmed.

Under zero-shot and few-shot conditions for every task, models were presented with the same prompt. Under zero-shot conditions, the model was given only the task description and had to come up with a full solution by itself without any further guidance. Under few-shot conditions, an example or several examples were used to guide the model's response. This framework allowed for testing how contextual signals influence model performance, as well as the extent of prior knowledge or scaffolding required for good output. Further, various versions of prompts—reparsed or restyled—were utilized to evaluate models' sensitivity to prompt wording, an attribute which is long documented to drastically affect LLM output.

Code produced by each model was rated with several evaluation metrics. Overall correctness was the chief metric and was ascertained by exercising the code using standard test cases within each benchmark suite. The ratio of test cases passed was a direct measure of semantic correctness. Syntactic correctness was determined by testing if the code would compile or run without syntax errors, a particularly significant consideration in statically typed languages. Style and readability were also qualitatively assessed using guidelines like variable naming, indentation, and conformance to standard programming practices.

Aside from direct execution-based metrics, model performance was also examined via code similarity scores such as BLEU (Bilingual Evaluation Understudy) and Levenshtein distance, which provide information on how well generated code resembles reference implementations. Although these scores are not always perfectly linear with functionality, they serve as useful secondary measures for comparative assessment.

Ethical concerns were also incorporated into the methodology. Measures were taken to make sure that none of the prompts or benchmarks had any proprietary or sensitive code, and the outputs were inspected for unintentional copying from public repositories, particularly in the case of big models trained on internet-scale datasets. Additionally, the evaluation process took into account the possible biases in generated code, including the employment of gendered variable names or stereotypical function labels.

This methodological framework gives a holistic and rigorous foundation for evaluating LLM capabilities in computer-aided software development. Through the use of controlled experiments along with varied benchmarks and multi-dimensional evaluation criteria, the study seeks to provide strong insights into both the capabilities and limitations of generation AI tools in the area of programming.

Results

The experimental analysis provided rich insights into the performance of Large Language Models (LLMs) in code generation on various fronts, such as functional correctness, syntactic correctness, language support, and response sensitivity to variations in prompts. This section discusses the results based on HumanEval, MBPP, and CodeXGLUE benchmarks, with an emphasis on Codex, CodeT5, and PLBART.

Functional correctness, measured through execution-based evaluation against pre-defined unit tests, exhibited significant variation across models and programming languages. Codex was always superior to other models, with an average pass rate of 78.5% across all the benchmarks in Python tasks. CodeT5 averaged 65.2%, while PLBART averaged 61.7%. The better performance of Codex is due to its enormous training corpus that contains large amounts of open-source code, as well as its autoregressive architecture that

has higher contextual coherence. Codex performed particularly well in data manipulation and basic algorithms like sorting, searching, and recursion. It tended to generate solutions that were not only functionally correct but also followed human coding conventions in terms of structure and readability.

Syntactic correctness was another domain where LLMs proved to be quite capable. More than 90% of outputs from all models were syntax-free, with Codex scoring 98.1%, CodeT5 94.3%, and PLBART 92.7%. Even when the models got the problem wrong from a functional perspective, they hardly ever generated malformed or non-compilable code. This indicates that LLMs possess strong internal grammar representations of programming languages, particularly those on which they have been heavily trained.

On code readability and stylistic quality, qualitative evaluations indicated that Codex-produced code generally adhered to standard naming conventions, employed uniform indentation, and had logical control structures. Still, certain outputs, especially those from PLBART, sometimes contained redundant code or less obvious naming conventions, suggesting a weakness in matching output with human-style stylistic biases. In addition, scoping and reuse of variables were not necessarily best in all cases, especially for models featuring encoder-decoder architecture, whose context retention among function components is often limited.

As for prompt sensitivity, findings highlighted the degree to which changing the wording of prompts affects the quality of the output. Codex had a fairly tolerant response to revised prompts, remaining functional correct for 84% of rewordings. CodeT5 and PLBART were more variable, with accuracy decreasing by 10–15% in rephrased conditions. This suggests that prompt engineering is still a vital component of using LLMs for software coding and points to the models' reliance on implicit contextual cues within the prompt structure.

Performance on various programming languages was another key dimension. Python was obviously the best-supported language out of all the models, probably because it is most common in training sets. The average accuracy of Codex fell from 78.5% in Python to 68.3% in JavaScript and 64.2% in Java. CodeT5 and PLBART followed the same pattern, with the worst performance seen for Java-based tasks. This indicates that although LLMs can generalize across languages, their performance is not even and is largely determined by language-specific representation during training.

As far as error types go, the majority of functional failures were due to logical misinterpretations of the task and not syntax-related problems. For example, some models would produce an off-by-one error in loop boundaries, misunderstand input constraints, or use incorrect data structures for the problem at hand. Such errors indicate the models' lack of abstract reasoning ability and exact algorithmic knowledge, particularly in problems involving several dependent steps.

Another interesting finding was code hallucination, wherein models produced non-existent functions or misremembered library imports. Although such events were comparatively rare with Codex (happening in 4.2% of test cases), they were more common in CodeT5 and PLBART, especially when the task description was not detailed enough. This once again highlights the importance of human supervision when implementing LLMs in actual software development environments.

Overall, the findings illustrate that although LLMs have great potential in producing high-quality, working code, their abilities are not yet advanced enough to completely automate software development. They work best as supporting tools that enhance human productivity instead of substituting human developers. Codex is at the forefront of the current technology, but even it has limitations in subtle understanding and context retention, particularly in multi-step problem-solving and under vague prompts.

Discussion

The experimental results provide substantial evidence that Large Language Models (LLMs), particularly transformer-based architectures like Codex, are redefining the landscape of automated software development. While the quantitative results are valuable in understanding model capabilities, this section seeks to contextualize those findings, interpret their implications, and critically analyze both the strengths and limitations of LLMs as tools for code generation.

One of the most striking observations from the results is the evident performance lead of Codex over its peers. This lead is not only apparent in functional accuracy but also in syntactic accuracy, style compliance, and robustness to prompt variation. Codex gains from both its broad-scale pretraining on a broad corpus of web text and its fine-tuning on hand-crafted programming data, which makes it exceedingly effective at comprehension of code-related syntax, semantics, and conventions. But this performance advantage also gives rise to some larger questions regarding coding data centralization, proprietary model creation, and access. Black-boxing of business models such as Codex restricts transparency in training data and design of architecture, which prevents independent replication and scrutiny on the part of the research community.

The results also confirm that code generation performance is highly language-biased. Models routinely performed better on Python than on Java or JavaScript, which probably indicates the dominance of Python in training datasets. The bias towards a language suggests that developers in less-populated or domain-specific languages will not enjoy the same advantage. It also emphasizes the need for diversified and balanced datasets for future model training to maintain fair support for all programming languages.

The problem of prompt sensitivity seen in models generally—and particularly those apart from Codex—has significant practical deployment implications. Although LLMs might produce correct code, their results are heavily contingent on the wording used to define tasks. The effect, typically called "prompt engineering," is a new form of skillset requirement for implementers of such tools. In conventional programming, the problem is applying logic; with LLM programming, some of the problem now is writing prompts that the model can successfully execute. While this enables quick prototyping and experimentation, it also adds uncertainty. Subtle differences in wording of prompts can result in extremely different outcomes, making debugging and reproducibility challenging.

Another crucial aspect is the noticed disparity between syntactic and semantic correctness. Although the majority of models produce syntactically correct code, they often don't fully grasp the complete logic necessary for the problem. This is to say that LLMs are better suited to pattern discovery and imitation rather than actual problem-solving or deduction. In problems involving multi-step

reasoning, branch conditions, or nested logic, models tend to make subtle yet fatal errors. These logical fallacies expose the current limitations of LLMs as independent problem solvers and emphasize the ongoing necessity for human control in production environments.

Ethical and legal issues are also raised in debate about LLM-produced code. As indicated in the associated literature, publicly trained models can unwittingly recreate snippets of copyrighted code or insecure coding patterns [8]. Additionally, there exists a non-negligible potential for producing vulnerable code, i.e., SQL injection-vulnerable logic or incorrect memory handling, particularly in languages like C or C++. In the absence of appropriate protections, the inclusion of such models in production workflows might pose substantial threats.

The application of LLMs in classrooms and low-code/no-code environments also deserves to be highlighted. Their capacity to produce readable and executable code from minimal prompts can act as a learning tool for beginners. But this brings into question pedagogical issues related to over-reliance on machine-sourced solutions. If junior developers or students habitually rely on LLMs to generate code, they risk never developing a strong grasp of programming logic, design patterns, and debugging processes. There is a requirement for careful incorporation of these tools into curricula, with the need to continue active learning and critical thinking.

In software development for enterprises, LLMs provide significant advantages in boilerplate code automation, unit test generation, cross-language code translation, and API documentation. Yet, their inability to comprehend intricate domain logic and system-level dependencies makes them inappropriate—at least currently—for mission-critical applications. Products such as GitHub Copilot are already affecting workflows, but the majority of practitioners concur that these models best serve as pair programmers or smart assistants, not as independent agents.

While LLMs have demonstrated impressive capacity in coded code generation, they are no magic bullet. Their application should be supported by a definitive comprehension of their benefits, such as quick prototyping and syntactic compliance, as well as their limitations, such as limits in logical reasoning and prompt structure sensitivity. As the domain matures, research is required in directions such as ongoing fine-tuning, human-in-the-loop designs, explainability, and ethical use in order to better leverage the promise of LLMs in software development.

Conclusion

The emergence of Large Language Models (LLMs) represents a watershed for automated software development. Codex and its equivalents such as CodeT5 and PLBART, in particular, have proven capable of producing syntactically acceptable, semantically relevant, and in some cases, production-quality code from natural language descriptions. The research findings indicate that LLMs have progressed considerably in comprehending the syntax and semantics of programming languages such that they can add value to various tasks from code completion and bug detection to complete function implementation. Yet, the evidence further presents the existing limitations hindering LLMs from achieving complete autonomy as software developers.

Perhaps the most remarkable result of the study is the frequency with which Codex bested its competition across all the benchmarks. With good performance on HumanEval and MBPP datasets, Codex was especially resilient on Python-based tasks, producing code that consistently passed test cases and followed best practices in coding. This implies that LLMs are not simply memorizing patterns but also acquiring generalized coding principles. However, the performance decline when switching to less often trained-on languages such as Java or JavaScript underscores the reliance of these models on the quality and distribution of their training data.

One of the key takeaways from the research is the significance of prompt formulation. Even the most powerful models, like Codex, show significant sensitivity to prompt phrasing variation. This implies that although LLMs can deliver correct outputs, they are strongly context-dependent, frequently needing iterative improvement and human intervention to deliver optimal results. This dependency on well-designed prompts suggests that LLMs are not yet at the level of independently comprehending software requirements as human developers, thus cementing their position as support tools instead of independent agents.

Additionally, syntactically correctness in all models is still astonishingly high, but semantic accuracy, logical consistency, and task-oriented reasoning issues still exist. Most code generation errors are a result of the models' misinterpretation of constraints or their failure to consistently perform multi-step logic. Such a limitation becomes especially problematic for applications that require safety-critical systems or security-sensitive code. Without thorough error detection, runtime enforcement, and human inspection, the possibility exists for buggy or risky code to make its way into production systems.

Another issue is the wide socio-technical effect of LLMs across software development. LLMs can potentially level the playing field for coding through reduced barriers of entry for novice coders, increased speed in professional development cycles, and augmenting learning platforms for students. But they also pose ethical issues in terms of intellectual property, plagiarism, model bias, and overdependence. LLM-generated code can unwittingly replicate proprietary or insecure logic from their training. In addition, the transparency of model training pipelines and uninterpretable nature contribute to accountability and trust issues for AI-assisted development tools.

Future work needs to concentrate on increasing model transparency, enhancing reasoning ability, and increasing multilingual and cross-domain generalization. Symbolic reasoning or constraint solvers, the use of reinforcement learning with human feedback, and the development of improved benchmarking tools can be used to address current limitations. In addition, cooperation among academia, industry, and open-source communities will be crucial in developing ethical, legal, and technical standards for the deployment of LLMs in software engineering applications.

Despite the fact that LLMs such as Codex are not yet ready to supplant human developers, they unequivocally provide a strong augmentation layer for contemporary software development. The potential of their capacity to automate repetitive coding chores, help with documentation, and accelerate prototyping is enormous. To actually fulfill their potential, however, will need a sustained emphasis on ensuring these models get aligned with human-centric development practices, effective evaluation frameworks, and accountable deployment strategies. As the discipline matures, the relationship between artificial intelligence and software engineering is poised to redefine not just how code is written, but how computer programmers engage with computing systems in general.

References

1. Vaswani A, Shazeer N, Parmar N (2017) Attention is all you need. Proc Adv Neural Inf Process Syst (NeurIPS) https://arxiv.org/abs/1706.03762.
2. Tom BB, Benjamin M, Nick R, Melanie S, Jared K, et al. (2020) Language Models are Few-Shot Learners. arXiv preprint https://arxiv.org/abs/2005.14165.
3. Daoguang Z, Bei C, Fengji Z, Dianjie L, Bingchao W, et al. (2022) Large Language Models Meet NL2Code: A Survey. arXiv preprint https://arxiv.org/abs/2212.09420.
4. Ribeiro R, Santos M, Silva J (2022) Evaluating Code Generation with AI Tools. Proc. Int. Conf. on Artificial Intelligence.
5. Zhang Z, Wang Y, Sun Q (2022) Code Generation with Large Language Models: Evaluation and Challenges. IEEE Trans Softw Eng 48: 3456-3470.
6. Juyong J, Fan W, Jiasi S, Sungju K, Sunghun K (2022) A Survey on Large Language Models for Code Generation. arXiv preprint https://arxiv.org/abs/2406.00515.
7. Zezhou Y, Sirong C, Cuiyun G, Zhenhao L, Ge L, et al. (2022) Deep Learning Based Code Generation Methods: Literature Review. arXiv preprint https://arxiv.org/abs/2303.01056.
8. Chaudhary S, Kumar A, Srinivas MTV (2021) AI-Powered Code Generation: A Review of Techniques and Tools. J of Software Engineering Research & Development 19: 201-215.