JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

ASSESSING CODE SYNTHESIS IN LLMs: CODEGEEX AND CODEIUM

[1]Saurabh Sheoran,[2]Dr.Dinesh Kumar
[1]M.Tech. Scholar,[2]Professor
Computer Science & Engineering
BRCM College of Engineering and Technology, Bahal, Bhiwani, Haryana, India

Abstract : Large Language Models (LLMs) have revolutionized software development in recent years, enabling the generation of high-quality source code across various domains. As the landscape of LLMs continues to evolve, developers are faced with the daunting task of selecting the most suitable model for their specific needs, with limited research providing practical guidance on this front. To address this challenge, this paper presents a comprehensive evaluation of two prominent LLMs, CodeGeeX and Codeium, on a diverse set of 25 programming tasks in Python and Java. The assessment encompasses both functional and non-functional aspects of the synthesized code, including performance, reliability, and maintainability, as well as developer usability and readability. The results show that while both models produced effective code, Codeium outperformed CodeGeeX in terms of code quality, clarity, and overall usability, with human assessor confirm these findings through manual reviews. This study provides valuable insights for developer seeking to harness the potential of LLMs in their software development workflows and high-lights the importance of evaluation and comparison of these models to ensure optimal results.

IndexTerms **-CodeGenerator, CodeGeeX, Codeium, developers, software development**

I. INTRODUCTION

Large Language Models (LLMs) are sophisticated AI systems designed to comprehend, produce and engage with human language, leveraging deep learning methods, particularly the transformer architecture, and trained on vast datasets comprising text from various sources, including books, websites, articles, and more. These models boast millions to trillions of parameters, enabling them to learn intricate patterns, grammar, context, and factual knowledge, thereby handling a broad range of natural language processing (NLP) tasks, such as text generation, summarization, translation, question answering, code generation, and conversation, often with minimal need for specialized training for each task. As a result, LLMs have gained widespread popularity and are being used for a variety of applications [9], including language translation, text summarization, code generation and chatbot.

The rapid adoption of LLMs in software engineering has been particularly notable, thanks to their advanced natural language processing capabilities. They are now widely used for tasks such as understanding and generating source code, with source code generation being the automated process of producing code from high level abstractions, typically defined in domain-specific languages or other declarative languages [3]. As more powerful LLMs emerge, developers face the challenge of selecting the most suitable model for their specific needs, requiring careful evaluation of the model's ability to produce high quality, readable, and maintainable code. It is essential to assess whether a model can generate code that is not only functional but also efficient, scalable, and secure, and whether developers can effectively integrate and adapt that code in real world applications, ensuring seamless collaboration between humans and machines.

While the benefits of saving time and reducing repetitive coding through automated source code generation are undeniable, the quality of the generated code remains a critical factor in software development. Code quality is a multifaceted concept that encompasses various aspects, including maintainability, readability, efficiency, and reliability. To evaluate code quality, developers often rely on methods like static code analysis, which examines the code without executing it. However, this approach has its limitations, as it may not always capture the full picture of code quality. Human judgment and expertise play a significant role in assessing code quality, as they can identify subtle issues and nuances that automated tools may miss. Therefore, it is essential to strike a balance between automated code analysis and manual assessment to ensure that generated source code meets the highest standards of quality and reliability.

Selecting the best Large Language Model (LLM) for a specific software engineering task is a significant challenge. While individual LLMs have been evaluated, comparing multiple models requires substantial effort and human involvement. This paper highlights a methodology [1] for assessing the quality of LLM generated code and evaluate two popular models - CodeGeeX and Codeium, across several quality dimensions. The study evaluates the models' ability to handle natural language instructions and the quality of the generated code in Python and Java.

The paper aims to answer three key questions:

1. How well do large language models generate high-quality code?
2. Is the code produced by the model considered usable by developers?
3. What key considerations influence developers' choice of LLM based coding tools?

The paper concludes by discussing related works, evaluating two large language models (LLMs) and highlighting the methodology. As these LLM are subject to frequent changes, one can evaluate these by using this methodology.

II. LITERATURE REVIEW

Zoltán Ságodi et al. [1] conducted a comparative evaluation of GitHub Copilot and ChatGPT for Java and C++ code generation, using 25 programming tasks [2] to assess functional and non-functional qualities. The study found that while both models produced high-quality code, ChatGPT outperformed Copilot overall, as confirmed by human reviewers. Thomas Helmuth et al. [2] introduced Program Synthesis Benchmark Suite 2, comprising 25 challenging problems sourced from katas and coursework. The suite is designed to be more difficult than its predecessor, with increased complexity demonstrated using the PushGP synthesis system.

Mark Chen, Jerry Tworek et al. [4] introduce Codex, a GPT model fine-tuned on GitHub code, demonstrating strong Python code generation capabilities and achieving high accuracy on the HumanEval benchmark through repeated sampling. P. Black [5] emphasizes that static analysis is essential for ethical software development, advocating its early use to detect vulnerabilities and encourage secure coding practices. Despite the shift to safer programming languages, static analysis remains crucial for verifying annotations and enforcing specifications. C. Kaner and W. P. Bond et al. [6] proposed a framework for evaluating software metrics, highlighting the limitations of bug counts and advocating for multidimensional analysis. They stress that code quality involves human judgment, which standard metrics often fail to reflect.

N. Al Madi [8] examined GitHub Copilot's code generation, comparing its readability and complexity to human-written code through a study with 21 participants. The results showed Copilot's code is similar to human generated code, but eye-tracking data revealed less visual attention on model generated code, which may affect code review and quality assurance. Diego Marcilio [11] et al. conducted a study on SonarQube usage across 246 projects, revealing that while automated static analysis tool warnings are valued, only 13% of issues were resolved, possibly due to inflated technical debt from excessive checkers. Despite this, the median issue resolution time was relatively quick at 18.99 days. D. Sobania [14] et al. evaluated GitHub Copilot against genetic programming approaches on program synthesis benchmarks, finding similar performance but concluding that genetic programming is not yet mature enough for practical software development due to its high training costs and slow solution generation.

N. Jain et al. [15] introduce Jigsaw, a tool that enhances large language models for code synthesis by incorporating post-processing steps based on program analysis, user feedback, and understanding of code syntax and semantics, improving accuracy over time. H. Pearce et al. [16] explored using large language models for zero-shot vulnerability repair in code, finding success in synthetic and hand-crafted cases but highlighting challenges in producing functionally correct code for real-world examples due to prompt design difficulties. Mohammed Latif Siddiq et al. [18] investigated the prevalence of code smells and security vulnerabilities in training datasets for code generation, finding 264 types of code smells and 44 types of security smells in popular datasets like CodeXGlue, APPS, and Code Clippy. Qinkai Zheng et al. [19] introduced CodeGeeX, a multilingual code generation model with 13 billion parameters, outperforming similar models in code generation and translation tasks, as evaluated by the HumanEval-X benchmark across multiple languages.

J. White et al. [21] introduce prompt patterns for automating software engineering tasks with large language models (LLMs), providing a catalog of solutions and exploring patterns to improve requirements elicitation, rapid prototyping, code quality, deployment, and testing. Arohi Srivastava et al. [31] introduced BIG-bench, a comprehensive, large-scale framework for evaluating large language models (LLMs), aiming to provide a more detailed and quantitative assessment beyond the traditional Turing Test. Touvron et al. [32] introduced LLaMA (Large Language Model Meta AI), an open, efficient, and accessible series of foundation models designed to democratize advanced NLP research and provide broader access to cutting-edge language model technology.

Ipek Ozkaya et al. [37] critically assess the role of LLMs in software engineering, weighing their benefits against limitations. They question whether models like GPT-3 and GPT-4 mark a true transformation in software development or if their impact is overhyped. Mari Ashiga et al. [53] survey the application of ensemble learning with LLMs for text and code generation, highlighting its potential to mitigate common issues like bias, inconsistency, and lack of diversity. They present ensemble methods as a promising way to enhance the reliability and quality of LLM outputs. Juyong Jiang et al. [52] present a comprehensive survey of recent progress in using LLMs for code generation, transformation and comprehension. They consolidate key research, tools, and methods, offering a valuable reference amid rapid advancements driven by models like Codex, PaLM-Coder and AlphaCode.

Belzner et al. [43] analyzed the growing impact of LLMs in software engineering, highlighting both their potential and limitations through a real-world case study. Zheng [41] et al. provided a comprehensive survey on the role of LLMs in code-related tasks, highlighting their evolution, benchmarking practices, and impact on software engineering.

III. METHODOLOGY

This section highlights a methodology [1] for comparing Large Language Models (LLMs) in the context of source code generation, emphasizing the crucial factors to consider when evaluating these models. The process / methodology adopted in this research or study is based on the process outlined / described by Zoltán Ságodi et al. (2024) [1], whose work is licensed under a Creative Commons Attribution–NonCommercial–NoDerivatives (CC BY-NC-ND) license.

The comparison process is structured into four distinct phases, each playing a vital role in assessing the models' performance: design the appropriate prompt to initiate the code generation process, code testing and verification of the generated code to ensure it meets the desired requirements, assessing its code quality to evaluate its maintainability, readability, and efficiency and conducting a manual assessment of the generated code to gauge the code's acceptability and usability.

In this study / research, the program synthesis benchmark [2] was used, that consisting of 25 diverse programming tasks drawn from multiple datasets. The program synthesis benchmark (PSB2) adopted in this research is outlined / described by Thomas Helmuth et al. (2021) [2], whose work is licensed under CC BY-NC-ND 4.0.

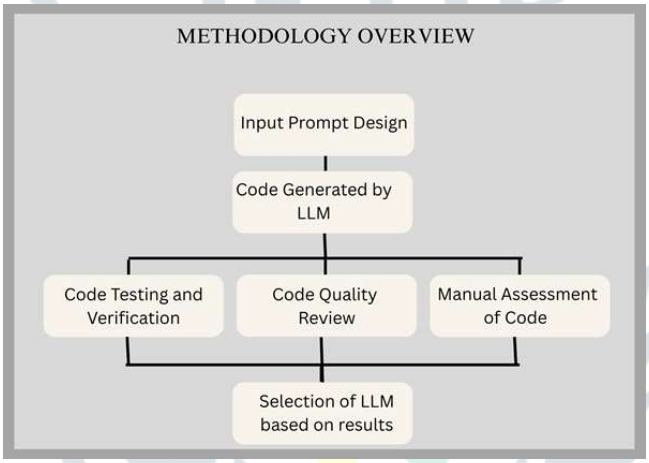

Figure 1: overview of the methodology

A. Input Prompt Design

In the realm of Large Language Models (LLMs), the art of prompt design is a critical component in guiding the model to produce desired outputs, particularly in code generation tasks. A well-crafted prompt serves as a blueprint, enabling the model to comprehend the task and generate accurate, relevant results. To achieve this, a prompt should be meticulously designed to be clear, specific, and detailed, encompassing key aspects of the task and accounting for potential edge cases or unexpected inputs.

Consider the task of generating a Python function to reverse a string. A well-constructed prompt can make all the difference in achieving the desired outcome.

Task: Write a Python function to reverse a string.
Prompt: "Write a Python function called ‘Reverse String' that takes a string as input and returns the string in reverse order."

In this example, the prompt is carefully crafted to provide explicit guidance, defining the function's name, input and expected output. The precise wording ensures that the model understands the task requirements, increasing the likelihood of generating correct and relevant code.

B. Code Testing and Verification

Code Testing and Verification is a crucial step in the code generation process using Large Language Models (LLMs). It involves assessing the generated code to ensure it functions as intended and solves the problem it was designed to address. In the context of LLM generated code, functional validity is evaluated by checking whether the code produces the correct outputs for the provided inputs.

To ensure the functional validity of LLM generated code, the following methods can be employed:

Execute tests on the generated code to verify its correctness and ensure it produces the expected outputs for a given set of inputs. Include a variety of test cases to demonstrate the solution's robustness and ability to handle different scenarios, and unexpected inputs. By verifying functional validity, developers can ensure that the LLM generated code is reliable, efficient, and effective in solving the intended problem, thereby increasing confidence in the code's quality and functionality.

C. Code Quality Review

Code Quality Review is a crucial step in evaluating the code generated by Large Language Models (LLMs). This process involves examining the technical accuracy, efficiency, and overall quality of the code to ensure it meets the required standards. When using LLMs for code generation, it is essential to verify that the produced code is not only syntactically correct but also semantically correct, as the model may generate code that is technically correct but logically flawed.

To guarantee the code quality of LLM generated code, it is crucial to verify that the code adheres to the correct syntax for the relevant programming language. This includes ensuring proper indentation, correct usage of functions, classes, variables, and control structures such as loops. Additionally, static analysis tools like CheckStyle and Pylint can be employed to catch syntax, style and other issues before runtime, ensuring that the code is technically sound and efficient. By assessing code technical quality, developers can ensure that the LLM generated code is reliable, maintainable, and efficient, reducing the risk of errors and improving overall code quality.

D. Manual Assessment

In the final phase of evaluating Large Language Model (LLM) generated code, a manual assessment, often a domain expert assesses the output to ensure it meets the required standards for correctness, quality, clarity, and efficiency. This step is crucial because, while LLMs can produce syntactically correct code, they may not fully grasp the deeper nuances of the problem, handle edge cases properly, or adhere to optimal coding practices. Manual assessment provides a subjective assessment of code quality, which may not be fully captured by traditional metrics.

Many related studies have overlooked the importance of manual assessment in assessing LLM generated code. While some studies have evaluated program synthesis, they have focused primarily on functional outcomes and user experience, without considering code quality, security, or performance. Others have examined static properties of the code, but lacked a detailed exploration of functional aspects and the prompting approach used in iterative development. This highlights the need for a methodology to effectively compare source code synthesis methods and ensure that LLM generated code meets the required standards.

IV. RESULT AND EXPERIMENT

This section presents a comparative analysis of two Large Language Models (LLMs) - CodeGeeX and Codeium, using the methodology outlined in section III above. This study provides a simplified example of a real-world comparison, illustrating the process for evaluating actual models. The comparison assesses the performance of both LLMs in generating high-quality code, highlighting their strengths and weaknesses. While this study provides a comprehensive evaluation, it is essential to note that a thorough evaluation should be conducted in a real-world environment, where resources such as high-performance server infrastructure and static code analysis tools are available, to obtain more accurate and reliable results.

A. Setup for Experiment

In the setup of this study, several factors were considered, including the choice of programming language and the selection of the large language model (LLM) for comparison.

1. ***Language Model Configuration for experiment:***

   CodeGeeX and Codeium were chosen for their capabilities and relevance to the task at hand. The study began with the installation of the respective extensions for both LLMs within Visual Studio Code, a popular Integrated Development Environment (IDE). Both LLMs feature an integrated chat window within Visual Studio Code, allowing developers to input specific queries or request code generation for a particular problem. I used these LLMs for code generation in first quarter of 2025.

2. ***Use of Programming Language:***

   For this study, I used both Python and Java programming languages. Python was chosen for its simplicity, extensive libraries, and robust community support, making it an ideal choice for quick development and experimentation. Java, on the other hand, is a prevalent programming language used in enterprise software development, making it an essential language for evaluating LLMs. By testing LLMs' ability to generate Java code, the study assesses how well these models perform in real-world software projects. The study uses both Python and Java to provide a more comprehensive assessment of how well LLMs can support various types of coding tasks across both research and industry contexts. This dual language approach offers valuable insights into the adaptability and dependability of LLMs when tackling a variety of programming tasks.

B. Prompt Design in LLM-Based Code Generation

Designing an appropriate prompt is crucial when using Large Language Models (LLMs) for automatic code generation. A well crafted prompt clearly communicates the intended programming task to the model, guiding it toward generating accurate, relevant, and useful code. Conversely, poorly structured prompts can lead to incorrect or inefficient code output. Effective prompts typically include essential information such as the target programming language and a concise yet informative description of the task. This structure helps the LLM to better understand the context and constraints of the problem, resulting in more accurate and readable code output.

In this study, a standardized prompt format was used to ensure clarity and neutrality between different LLMs. The prompt format consisted of a fixed prefix, "Write a '<LANGUAGE>' code to solve the task" followed by the task description. This uniform format helped to maintain simplicity and balance, avoiding bias and ensuring a fair comparison of the LLMs' code generation capabilities. In this study, I used a program synthesis benchmark [2] consisting of 25 diverse programming tasks drawn from multiple datasets.

It is essential to keep prompts simple and balanced to avoid introducing bias and undermining the fairness and consistency of the evaluation. By maintaining prompt simplicity and neutrality, this study ensured that both LLMs were tested under equal conditions, enabling a more reliable comparison of their code generation capabilities.

C. Code Testing and Verification

Code testing and verification in automatic code generation using Large Language Models (LLMs) involves evaluating whether the generated code executes correctly and produces the expected results for a given task. While syntactic correctness ensures that the code is free of compilation or runtime errors, functional validity goes a step further by verifying the code's logical accuracy - whether it actually fulfills the intended functionality as described in the prompt.

This assessment typically involves running the generated code and check with specific input values and corresponding expected outputs. If the outputs produced by the code match the expected results across all or most of these test cases, the code is considered functionally valid. In this experiment / case study, the number of inputs used for a program / task was less than or equal to 10. This method mirrors standard software testing practices used in real world development environments and is critical for determining whether the code not only compiles but also performs the correct operations.

Code testing and verification is a key metric in evaluating LLM generated code because it reflects the model's ability to understand and implement the underlying logic of a task, rather than just producing code that "looks right". Even minor logical errors - such as indexing error, incorrect conditions or improper handling of edge cases can render code unusable in practical applications. In enterprise or production settings, such errors can lead to serious bugs, application crashes, or security vulnerabilities.

Therefore, incorporating functional validity into the evaluation process helps bridge the gap between code generation in research and real-world software development. It ensures that the LLMs are not only producing clean and readable code but are also capable of solving problems accurately and reliably. To check the functional validity of the tasks, I executed all 25 benchmark tasks [2] with inputs as per benchmark. I executed each task with number of inputs up to 10. For some of the tasks there were few errors or incorrect output and rest all tasks gave expected result for given inputs.

1) Python Results

I conducted code testing and verification on all programs generated by both CodeGeeX and Codeium using various input values, executing each program and comparing its outputs against expected results to determine if it performed the intended functionality correctly. The comprehensive outcomes of this testing process, including results for each task and model, are presented in below table 1, which clearly compares the functional validity of Python code generated by both models across the entire set of programming tasks.

Upon reviewing the generated code for various tasks, I observed that some of the programs generated by CodeGeeX and Codeium produced different results for the same input values. CodeGeeX's code for task *bouncingBalls* gave incorrect results, whereas Codeium's code produced the correct output. Additionally, CodeGeeX's code for task *bowling* threw a ValueError for certain input values, while Codeium's code for task *camelCase* returned incorrect results when input was given as 'grouping of words separated by a space'. Both CodeGeeX and Codeium failed to handle negative input values properly for tasks *coinSums* and *diceGame*, returning incorrect results. Furthermore, CodeGeeX's code for task *findPair* returned the index of the value instead of the value itself.

Table 1: python: code testing and verification results

Task	CodeGeex	Codeium
Basement	100%	100%
BouncingBalls	0%	100%
Bowling	60%	100%
CamelCase	100%	60%
CoinSums	80%	80%
CutVector	100%	80%
DiceGame	80%	80%
FindPair	0%	100%
FizzBuzz	100%	100%
FuelCost	100%	100%
GCD	100%	100%
IndicesofSubstring	80%	100%
Leaders	80%	100%
Luhn	0%	100%
Mastermind	100%	100%
MiddleCharacter	100%	100%
PairedDigits	100%	100%
ShoppingList	0%	100%
SnowDay	80%	80%
SolveBoolean	100%	100%
SpinWords	100%	100%
SquareDigits	80%	100%
SubstitutionCipher	100%	100%
Twitter	100%	100%
VectorDistance	100%	100%

For CodeGeeX, the tasks *indicesOfSubstring* and *leaders* do not return a vector as output, and the program generated by CodeGeeX for the task *luhn* does not take a vector as input as expected. Regarding the task *shoppingList* by CodeGeeX, it produced incorrect results for given inputs, whereas the same program code generated by Codeium provided the correct results. Both CodeGeeX and Codeium fail to handle negative input values properly for the task *snowDay*, and CodeGeeX's code for *squareDigit* did not handle the native input correctly. Out of the 25 tasks in the benchmark, CodeGeeX generated code achieved 100% expected results for 14 tasks, while Codeium generated code achieved the same for 19 tasks.

2) Java Results

I tested all Java programs generated by both CodeGeeX and Codeium using a variety of input values. All results are available in table 2

Table 2: java: code testing and verification results

Task	CodeGeex	Codeium
Basement	100%	100%
BouncingBalls	0%	100%
Bowling	100%	0%
CamelCase	100%	100%
CoinSums	100%	100%
CutVector	80%	80%
DiceGame	100%	100%
FindPair	0%	100%
FizzBuzz	100%	100%
FuelCost	100%	100%
GCD	100%	100%
IndicesofSubstring	100%	100%
Leaders	100%	100%
Luhn	60%	100%
Mastermind	100%	100%
MiddleCharacter	100%	100%
PairedDigits	100%	100%
ShoppingList	0%	100%
SnowDay	100%	100%
SolveBoolean	80%	100%
SpinWords	100%	100%
SquareDigits	100%	100%
SubstitutionCipher	100%	100%
Twitter	100%	100%
VectorDistance	100%	100%

Out of the 25 tasks, CodeGeeX *bouncingBalls* produced incorrect output for the given input, while Codeium *bowling* task encountered an index out of range error for all input values. Both LLMs failed to handle negative input properly for the task *cutVector* and the task *findpair* returned the index of the value instead of the value itself in the case of CodeGeeX.

Upon reviewing the generated programs, I observed several discrepancies in their outputs. Specifically, the program generated by CodeGeeX for task *luhn* failed to accept a vector as input, contrary to expectations. Additionally, task *shoppingList* produced incorrect results with CodeGeeX whereas Codeium's generated code yielded correct outputs for the same input values. Furthermore, CodeGeeX's program for task *solveBoolean* returned incorrect results for certain input values, whereas Codeium's program produced correct outputs for the same inputs.

Out of the 25 tasks in the benchmark, the code generated by CodeGeeX produced 100% expected results for 19 tasks, while the code generated by Codeium achieved the same for 23 tasks.

D. Code Quality Review

Code Quality Review in auto code generation using Large Language Models (LLMs) involves evaluating whether the generated code adheres to correct programming rules and standards, ensuring it is free of syntax errors, employs proper structure, and follows best practices of the chosen programming language, such as Python or Java. This assessment guarantees that the code can be compiled or executed without issues and examines aspects like correct use of variables, functions, loops, and data types. Verifying code technical validity is crucial because, even if the code appears correct, it must be written in a manner that computers can understand and run smoothly without crashing.

In evaluating the static analysis results, I utilized multiple parameters including Code Smells, Number of Logical Statements, Count of Code Statements, Cyclomatic Complexity (CC), and Depth of Nested Code. For static code analysis, CheckStyle was used for Java programs, and Pylint was used for Python programs. However, for some parameters I did manual validation. Among these metrics, lower values are considered preferable.

1) *Python Results*

The static analysis results of Python code generated by both CodeGeeX and Codeium are presented in table 3. For Python, the code generated for all tasks by CodeGeeX is free of Code Smells, whereas there are two instances of Code Smells in Codeium generated code, in the tasks *mastermind* and *solveBoolean*. In terms of 'Number of Logical Statements' CodeGeeX has 7 tasks with a higher count than their corresponding Codeium generated programs, while Codeium has 4 tasks with a higher 'Number of Logical Statements' count than their corresponding CodeGeeX generated programs.

Table 3: python: code quality review results

Task	Code Smells		Number of Logical Statements		Count of Code Statements		Cyclomatic Complexity (CC)		Depth of Nested Code		Result (Combine all 5 parameters results count)	
	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium
Basement	0	0	7	7	7	7	3	3	2	2	5	5
BouncingBalls	0	0	10	9	10	9	2	2	1	1	3	5
Bowling	0	0	22	16	22	16	7	5	3	2	1	5
CamelCase	0	0	6	8	6	8	2	3	1	2	5	1
CoinSums	0	0	10	7	10	7	1	2	0	1	3	3
CutVector	0	0	13	13	13	13	3	3	2	2	5	5
DiceGame	0	0	8	8	8	8	4	4	3	3	5	5
FindPair	0	0	8	8	8	8	3	3	2	2	5	5
FizzBuzz	0	0	9	9	9	9	5	5	1	1	5	5
FuelCost	0	0	2	2	2	2	2	2	1	1	5	5
GCD	0	0	4	4	4	4	2	2	1	1	5	5
IndicesofSubstring	0	0	7	6	7	6	3	3	2	2	3	5
Leaders	0	0	9	8	9	8	3	3	2	2	3	5
Luhn	0	0	8	11	8	11	3	5	2	3	5	1
Mastermind	0	1	4	17	4	17	1	6	0	2	5	0
MiddleCharacter	0	0	6	7	6	7	3	3	1	1	5	3
PairedDigits	0	0	6	6	6	6	3	3	2	2	5	5
ShoppingList	0	0	5	6	5	6	2	2	1	1	5	3
SnowDay	0	0	6	6	6	6	2	2	1	1	5	5
SolveBoolean	0	1	13	4	13	4	6	1	1	0	1	4
SpinWords	0	0	4	4	4	4	4	4	2	2	5	5
SquareDigits	0	0	2	2	2	2	2	2	1	1	5	5
SubstitutionCipher	0	0	8	3	8	3	4	1	2	0	1	5
Twitter	0	0	7	7	7	7	4	4	1	1	5	5
VectorDistance	0	0	4	4	4	4	3	3	1	1	5	5

An analysis of the 'Count of Code Statements' parameter revealed that CodeGeeX generated code with more number of statements than Codeium for 7 tasks. Conversely, Codeium generated code with more number of statements than CodeGeeX for 4 tasks. This indicates that both CodeGeeX and Codeium have varying levels of verbosity in their generated code, with CodeGeeX producing more statements in some cases and Codeium producing more statements in others.

Regarding the Cyclomatic Complexity (CC) parameter, CodeGeeX has 3 tasks where the CC count is higher than their corresponding Codeium generated programs, while Codeium has 4 tasks with a higher CC count than their corresponding CodeGeeX generated programs. The difference in' Depth of Nested Code' between programs generated by CodeGeeX and Codeium is minimal.

A comprehensive analysis was conducted by combining the results of 'Code Smells', 'Number of Logical Statements', 'Count of Code Statements', Cyclomatic Complexity (CC), 'Depth of Nested Code' for each task, generated by CodeGeeX and Codeium separately. The combined results revealed that there is not a significant difference between the two LLMs. However, a closer examination showed that CodeGeeX outperformed Codeium in 5 tasks, while Codeium excelled in 6 tasks. This suggests that both LLMs have their strengths and weaknesses.

Approach for calculation of aforesaid final combined result: e.g. For the task *bouncingBalls*: both LLMs scored 0 on 'Code Smells'; CodeGeeX scored 10 and Codeium 9 on 'Number of Logical Statements'; CodeGeeX scored 10 and Codeium 9 on 'Count of Code Statements'; both scored 2 on Cyclomatic Complexity (CC); and both scored 1 on 'Depth of Nested Code'. Based on these metrics, the final combined score is 5 for Codeium and 3 for CodeGeeX. (because In the case of ties across three parameters, each LLM received 3 points / score, with 2 additional points / score awarded to Codeium)

2) Java Results

The results of the static analysis of Java code generated by CodeGeeX and Codeium are presented in table 4. The analysis reveals that CodeGeeX outperformed Codeium in terms of Code Smells, with CodeGeeX generated code having a lower count of Code Smells compared to Codeium generated code. Specifically, the maximum count of Code Smells in Codeium generated code was 15 for the task bowling, whereas the corresponding task in CodeGeeX generated code had a Code Smells count of 8 (highest for CodeGeeX).

For CodeGeeX, there are 7 tasks with a higher count of 'Number of Logical Statements' than their corresponding programs generated by Codeium. Conversely, Codeium has 8 tasks with a higher 'Number of Logical Statements' count than their corresponding CodeGeeX generated programs. The maximum 'Number of Logical Statements' count is 36 for the *bowling* task in Codeium generated code, while the corresponding same task in CodeGeeX has count 25 for 'Number of Logical Statements'.

An analysis of the 'Count of Code Statements' parameter revealed that CodeGeeX generated code with more statements than Codeium for 7 tasks while Codeium generated code with more statements than CodeGeeX for 10 tasks. The maximum count of 'Count of Code Statements' parameter was 34 for the task bowling in Codeium generated code, whereas the corresponding task in CodeGeeX generated code had a count 23 for 'Count of Code Statements' parameter.

For the Cyclomatic Complexity (CC) parameter, CodeGeeX has 2 tasks with a higher CC count than their corresponding programs generated by Codeium, while Codeium has 11 tasks with a higher CC count than their corresponding CodeGeeX generated programs. Although there is not much difference in the 'Depth of Nested Code' between programs generated by CodeGeeX and Codeium, overall, CodeGeeX generated programs perform better when compared to Codeium generated programs for the CC parameter.

A comprehensive analysis was conducted by combining the results of 'Code Smells', 'Number of Logical Statements', 'Count of Code Statements', Cyclomatic Complexity (CC) and 'Depth of Nested Code' for each task generated by CodeGeeX and Codeium separately. The combined results revealed that CodeGeeX outperformed Codeium in 12 tasks, while Codeium excelled in 7 tasks. Overall, the analysis suggests that CodeGeeX performed better than Codeium in terms of code quality of Java code, with a higher number of tasks meeting the desired standards.

Approach for calculation of aforesaid final combined result: e.g. For the task *bouncingBalls*: CodeGeeX scored 7 and Codeium 9 on 'Code Smells'; CodeGeeX scored 10 and Codeium 9 on 'Number of Logical Statements'; CodeGeeX scored 7 and Codeium 6 on 'Count of Code Statements'; both scored 2 on Cyclomatic Complexity (CC); and both scored 1 on 'Depth of Nested Code'. Based on these metrics, the final combined score is 4 for Codeium and 3 for CodeGeeX. (because In the case of ties across two parameters, each LLM received 2 points / score, with 1 additional points / score awarded to CodeGeeX and 2 additional points / score awarded to Codeium)

Table 4: java: code quality review results

	Code Smells		Number of Logical Statements		Count of Code Statements		Cyclomatic Complexity (CC)		Depth of Nested Code		Result (Combine all 5 parameters results count)	
Task	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium
Basement	1	1	8	8	6	6	3	3	2	2	5	5
BouncingBalls	7	9	10	9	7	6	2	2	1	1	3	4
Bowling	8	15	25	36	23	34	10	17	3	5	5	0
CamelCase	2	3	9	12	7	10	2	3	1	2	5	0
CoinSums	7	7	11	10	9	8	1	1	0	0	3	5
CutVector	1	3	25	20	23	17	6	4	2	2	2	4
DiceGame	3	2	12	9	8	7	4	4	3	3	2	5
FindPair	3	2	11	14	7	12	3	5	2	3	4	1
FizzBuzz	4	2	10	10	8	8	5	5	1	1	4	5
FuelCost	2	2	6	6	4	4	2	2	1	1	5	5
GCD	2	2	7	6	5	4	2	3	1	1	3	4
IndicesofSubstring	2	2	10	9	6	5	3	3	2	2	3	5
Leaders	1	1	15	12	11	8	2	3	1	2	3	3
Luhn	3	3	12	12	10	10	4	5	3	3	5	4
Mastermind	3	6	21	21	17	19	6	8	2	3	5	1
MiddleCharacter	1	1	7	8	5	6	3	3	1	1	5	3
PairedDigits	1	1	7	7	5	5	3	3	2	2	5	5
ShoppingList	3	4	6	7	4	5	2	2	1	1	5	2
SnowDay	5	5	7	7	5	5	2	2	1	1	5	5
SolveBoolean	4	4	14	28	11	24	6	10	1	2	5	1
SpinWords	5	2	7	11	5	9	3	4	2	2	4	2
SquareDigits	1	1	3	8	1	6	1	2	0	1	5	1
SubstitutionCipher	4	4	10	14	8	10	4	5	2	2	5	2
Twitter	2	2	8	8	6	6	4	4	1	1	5	5
VectorDistance	3	4	7	8	5	6	3	2	2	1	3	2

E. Manual Assessment

Manual Assessment plays a vital role in assessing the quality of code generated by Large Language Models (LLMs). Unlike automated testing, which mainly checks for functional correctness, manual assessment involves manual review by experienced developers or researchers to provide deeper insights into the code. In this study, a detailed manual assessment was carried out on code samples generated by LLMs in both Java and Python.

Evaluator reviewed the code for various qualitative aspects, helping to identify strengths and weaknesses in the generated outputs across both programming languages.

The evaluation criteria included following important parameters, for each parameter, evaluator has to give score which is explained below.

i. Early observations about the code: Early observations about the code mean the first things you notice when you start looking at the code. This includes how it's organized, and how names are used etc before checking it in detail.

ii. Ease of Use: Ease of use means how simple it is for other developers to read and understand the code. If the code is neat, clear, and has helpful comments, it's easier to change or use again.

iii. Readability: Readability refers to how easily a developer can understand the code just by looking at it. Clear naming, proper formatting, and simple logic make the code easier to read and work with.

iv. Maintainability: Maintainability means how easy it is to read, change, or improve the code later. If the code is neat, clear, and well-structured, it's easier to work with in the future.

v. Clarity and Simplicity: Clarity and simplicity of code mean the code is easy to read and not too complex. Clear code makes it easier for you and others to understand what it does.

Each task was rated on a scale from 1 to 4, where 4 is very good, 3 is good, 2 is bad and 1 is very bad.

This structured approach enabled a subtle, human centered evaluation of LLM generated code, providing insights that surpass what automated tools can offer. After scoring all the values, the reviewer had to determine which of the presented source code was superior.

1) Python Results

For Python, as shown in table 5, Codeium outperforms CodeGeeX. The 'Early observations of code' scores for both CodeGeeX and Codeium are nearly identical, with 20 out of 25 tasks receiving the same score for both LLMs. However,

Codeium maintains a slight lead over CodeGeeX generated programs overall. Similarly, the 'Ease of Use' scores for both CodeGeeX and Codeium are comparable, with 20 tasks sharing the same score, yet Codeium remains marginally ahead of CodeGeeX generated programs.

An evaluation of the generated code revealed that Codeium outperformed CodeGeeX in terms of readability. Codeium generated code included comments, whereas CodeGeeX generated code lacked comments entirely. On the Maintainability scale, both Codeium and CodeGeeX were found to be almost equal. However, Codeium generated code excelled on the 'Clarity and Simplicity' scale, with its code has more 'clarity and simplicity' score than CodeGeeX generated code.

After combining the scores for all parameters – 'Early observations about the code', 'Ease of Use', Readability, Maintainability and 'Clarity and Simplicity' for each task of both CodeGeeX and Codeium separately and examining the combined results, it was found that only one task of CodeGeeX outperformed Codeium (with combined results of all 5 parameters score values) . In contrast, Codeium had better results than CodeGeeX for 23 tasks.

Both CodeGeeX and Codeium demonstrated their capabilities by generating good programs in the Python language. However, a closer examination of the results revealed small differences across the five evaluation parameters. These differences provide valuable insights into the strengths and weaknesses of each LLM. The human evaluation results for the Python code generated by both LLMs are summarized in below table 5, providing a comprehensive overview of their performance.

Table 5: python: manual assessment of code results

	Early observations about the code		Ease of Use		Readability		Maintainability		Clarity and Simplicity		Result	
Task	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium
Basement	4	4	4	4	3	4	4	4	4	4	19	20
BouncingBalls	3	4	2	4	3	4	4	4	2	4	14	20
Bowling	4	4	3	4	3	4	3	4	3	4	16	20
CamelCase	4	4	4	4	3	4	4	4	4	4	19	20
CoinSums	4	4	4	4	3	4	4	4	4	4	19	20
CutVector	4	4	4	4	3	4	4	4	4	4	19	20
DiceGame	4	4	4	4	3	4	4	4	4	4	19	20
FindPair	2	4	1	4	3	4	4	4	2	4	12	20
FizzBuzz	4	4	4	4	3	4	4	4	4	4	19	20
FuelCost	4	4	4	4	3	4	4	4	4	4	19	20
GCD	4	4	4	4	3	4	4	4	4	4	19	20
IndicesofSubstring	4	4	4	4	3	4	4	4	4	4	19	20
Leaders	4	4	4	4	3	4	4	4	4	4	19	20
Luhn	2	4	2	4	3	4	4	4	2	4	13	20
Mastermind	3	3	4	4	3	3	3	3	3	3	16	16
MiddleCharacter	4	4	4	4	3	4	4	4	4	4	19	20
PairedDigits	4	3	4	4	3	3	4	4	4	4	19	18
ShoppingList	3	4	2	4	3	4	4	4	4	4	16	20
SnowDay	4	4	3	3	3	4	4	4	3	3	17	18
SolveBoolean	4	4	4	4	3	4	4	4	4	4	19	20
SpinWords	4	4	4	4	3	4	4	4	4	4	19	20
SquareDigits	4	4	4	4	3	4	4	4	4	4	19	20
SubstitutionCipher	4	4	4	4	3	4	4	4	4	4	19	20
Twitter	4	4	4	4	3	4	4	4	4	4	19	20
VectorDistance	4	4	4	4	3	4	4	4	4	4	19	20

2) Java Results

For Java, the results in below table 6 show that Codeium performs better than CodeGeeX.

Table 6: java: manual assessment of code results

	Early observations about the code		Ease of Use		Readability		Maintainability		Clarity and Simplicity		Result	
Task	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium	CodeGeex	Codeium
Basement	4	4	4	4	3	3	4	4	4	4	19	19
BouncingBalls	1	4	2	4	3	3	4	4	1	4	11	19
Bowling	4	4	4	3	3	3	3	3	3	3	17	16
CamelCase	4	4	4	4	3	3	4	4	4	4	19	19
CoinSums	4	4	4	4	3	3	4	4	4	4	19	19
CutVector	4	4	4	4	3	3	4	4	4	4	19	19
DiceGame	4	4	4	4	3	3	4	4	4	4	19	19
FindPair	3	4	2	4	3	3	4	4	2	4	12	19
FizzBuzz	4	4	4	4	3	3	4	4	4	4	19	19
FuelCost	4	4	4	4	3	3	4	4	4	4	19	19
GCD	4	4	4	4	3	3	4	4	4	4	19	19
IndicesofSubstring	4	4	4	4	3	3	4	4	4	4	19	19
Leaders	4	4	4	4	3	3	4	4	4	4	19	19
Luhn	3	4	3	3	3	3	4	4	2	4	15	18
Mastermind	3	4	4	4	3	3	4	4	3	4	17	19
MiddleCharacter	4	4	4	4	3	3	4	4	4	4	19	19
PairedDigits	4	4	4	4	3	3	4	4	4	4	19	19
ShoppingList	1	4	1	4	3	3	4	4	1	4	10	19
SnowDay	4	4	4	4	3	3	4	4	4	4	19	19
SolveBoolean	3	4	3	4	3	3	4	4	3	4	16	19
SpinWords	4	4	4	4	3	3	4	4	4	4	19	19
SquareDigits	3	3	3	3	3	3	4	4	3	3	16	16
SubstitutionCipher	4	4	4	4	3	3	4	4	4	4	19	19
Twitter	4	4	4	4	3	3	4	4	4	4	19	19
VectorDistance	4	4	3	3	3	3	4	4	3	3	17	17

Both models have nearly identical 'Early observations of code' scores, with 19 out of 25 tasks showing the same rating. However, Codeium has a slight edge overall. Similarly, the 'Ease of Use' scores are also comparable, with 20 out of 25 tasks rated equally, but once again, Codeium slightly outperforms the programs generated by CodeGeeX.

In terms of readability, the Java code generated by Codeium is comparable to that generated by CodeGeeX. A score of 3 was awarded instead of 4 for all tasks due to the absence of comments in the programs produced by all LLMs. Regarding Maintainability, both CodeGeeX and Codeium are nearly at the same level. On the 'Clarity and Simplicity' scale, the Java code generated by Codeium slightly surpasses the code generated by CodeGeeX.

A comprehensive analysis was conducted by combining the scores of 'Early observations about the code', 'Ease of Use', Readability, Maintainability and 'Clarity and Simplicity' for each task generated by CodeGeeX and Codeium separately. The combined results revealed that only 1 task of CodeGeeX outperformed Codeium (with combined results of all 5 parameters score values), whereas Codeium excelled in 6 tasks.

3) Code Comments

Code comments are short notes written in a program to explain what the code is doing. They are not run by the computer but help people understand the code more easily. Comments are important because they make the code easier to read, fix, and work on with others. Code comments were present in Python programs only generated by Codeium. There were no comments present for Codeium Java programs and for CodeGeeX generated Java and Python programs. Below table 7 contains details about code comments for both CodeGeeX and Codeium generated programs.

Table 7: code comments

Java			Python		
	Code Comment			Code Comment	
Task	CodeGeex	Codeium	Task	CodeGeex	Codeium
Basement	No Comments	No Comments	Basement	No Comments	Comments Present
BouncingBalls	No Comments	No Comments	BouncingBalls	No Comments	Comments Present
Bowling	No Comments	No Comments	Bowling	No Comments	Comments Present
CamelCase	No Comments	No Comments	CamelCase	No Comments	Comments Present
CoinSums	No Comments	No Comments	CoinSums	No Comments	Comments Present
CutVector	No Comments	No Comments	CutVector	No Comments	Comments Present
DiceGame	No Comments	No Comments	DiceGame	No Comments	Comments Present
FindPair	No Comments	No Comments	FindPair	No Comments	Comments Present
FizzBuzz	No Comments	No Comments	FizzBuzz	No Comments	Comments Present
FuelCost	No Comments	No Comments	FuelCost	No Comments	Comments Present
GCD	No Comments	No Comments	GCD	No Comments	Comments Present
IndicesofSubstring	No Comments	No Comments	IndicesofSubstring	No Comments	Comments Present
Leaders	No Comments	No Comments	Leaders	No Comments	Comments Present
Luhn	No Comments	No Comments	Luhn	No Comments	Comments Present
Mastermind	No Comments	No Comments	Mastermind	No Comments	Comments Present
MiddleCharacter	No Comments	No Comments	MiddleCharacter	No Comments	Comments Present
PairedDigits	No Comments	No Comments	PairedDigits	No Comments	Comments Present
ShoppingList	No Comments	No Comments	ShoppingList	No Comments	Comments Present
SnowDay	No Comments	No Comments	SnowDay	No Comments	Comments Present
SolveBoolean	No Comments	No Comments	SolveBoolean	No Comments	Comments Present
SpinWords	No Comments	No Comments	SpinWords	No Comments	Comments Present
SquareDigits	No Comments	No Comments	SquareDigits	No Comments	Comments Present
SubstitutionCipher	No Comments	No Comments	SubstitutionCipher	No Comments	Comments Present
Twitter	No Comments	No Comments	Twitter	No Comments	Comments Present
VectorDistance	No Comments	No Comments	VectorDistance	No Comments	Comments Present

V. CONCLUSION AND FUTURE WORK

This case study highlights a methodology for evaluating the code generation abilities of Large Language Models (LLMs), also evaluated both CodeGeeX and Codeium. The study highlights the importance of detailed prompting and code testing and verification, and examines the strengths and weaknesses of both models. By applying this methodology (mentioned in aforesaid section III) to a benchmark of 25 tasks [2], the study found that Codeium performed better than CodeGeeX for both languages, particularly in generating Python code. While both models produced high quality code with some minor issues however Codeium is recommended for program synthesis tasks.

Future studies can build on this evaluation method by applying it to other programming languages and LLMs, helping developers select the best tool for code generation. As LLMs advance, it is essential to update and improve the evaluation framework to align with new model capabilities. Expanding the evaluation to include more programming languages, real-world development scenarios and user feedback can provide deeper insights into practical model performance. Incorporating collaborative coding scenarios and refining the evaluation metrics can ensure that selected LLMs meet both technical and usability standards in diverse software engineering environments.

REFERENCES

[1] Zoltán Ságodi,István Siket, Rudolf Ferenc, "Methodology for Code Synthesis Evaluation of LLMs Presented by a Case Study of ChatGPT and Copilot," IEEE Access, volume 12, pages 72303-72316, 2024

[2] Thomas Helmuth and Peter Kelly, "PSB2: The second program synthesis benchmark suite", Proc. Genetic Evol. Comput. Conf., pp. 10-14, Jun. 2021.

[3] A. Dieumegard, A. Toom, and M. Pantel, ''Model-based formal specification of a DSL library for a qualified code generator,'' in Proc. 12th Workshop OCL Textual Modeling New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 61–62, doi: 10.1145/2428516.2428527.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto et al., ''Evaluating large language models trained on code,'' 2021, arXiv:2107.03374.

[5] P. Black, ''Static analyzers: Seat belts for your code,'' IEEE Secur. Privacy, vol. 10, no. 3, pp. 48–52, May 2012.

[6] C. Kaner and W. P. Bond, ''Software engineering metrics: What do they measure and how do we know?'' in Proc. Int. Softw. Metrics Symp., Chicago, IL, USA, 2004, pp. 1–12.

[7] P. Vaithilingam, T. Zhang, and E. L. Glassman, ''Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models,'' in Proc. Extended Abstr. CHI Conf. Hum. Factors Comput. Syst. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 1–7, doi: 10.1145/3491101.3519665.

[8] N. Al Madi, ''How readable is model-generated code? Examining readability and visual inspection of Github copilot,'' in Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng. New York, NY, USA: Association for Computing Machinery, Feb. 2023, pp. 1–5, doi: 10.1145/3551349.3560438.

[9] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz et al. (Mar. 2023). Sparks of Artificial General Intelligence: Early Experiments With GPT-4. March 2023. [Online]. https://arxiv.org/abs/2303.12712

[10] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, ''Automatically generating fix suggestions in response to static code analysis warnings,'' in Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation (SCAM), Sep. 2019, pp. 34–44.

[11] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz et al. ''Are static analysis violations really fixed? A closer look at realistic usage of SonarQube,'' in Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC), May 2019, pp. 209–219.

[12] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, and N. Meng, ''Example based vulnerability detection and repair in Java code,'' in Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC). New York, NY, USA: Association for Computing Machinery, May 2022, pp. 190–201, doi: 10.1145/3524610.3527749.

[13] C. Maddison and D. Tarlow, ''Structured generative models of natural source code,'' in Proc. Int. Conf. Mach. Learn., 2014, pp. 649–657.

[14] Dominik Sobania, M. Briesch, and F. Rothlauf, ''Choose your programming copilot: A comparison of the program synthesis performance of GitHub copilot and genetic programming,'' in Proc. Genetic Evol. Comput. Conf. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1019–1027, doi: 10.1145/3512290.3528700.

[15] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy et al. ''Jigsaw: Large language models meet program synthesis,'' in Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE). New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1219–1231, doi: 10.1145/3510003.3510203.

[16] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, ''Examining zero-shot vulnerability repair with large language models,'' in Proc. IEEE Symp. Secur. Privacy (SP), May 2023, pp. 2339–2356.

[17] O. Asare, M. Nagappan, and N. Asokan, ''Is Github's copilot as bad as humans at introducing vulnerabilities in code?'' Empirical Softw. Eng., vol. 28, p. 129, Jan. 2023.

[18] 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 Proc. IEEE 22nd Int. Work. Conf. Source Code Anal. Manipulation (SCAM), Oct. 2022, pp. 71–82.

[19] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang et al., "CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X" , DOI:10.48550/arXiv.2303.17568, March 2023

[20] Latha Ramamoorthy, "AI for Software Engineering -Enhancing Developer Experience with Codeium and Copilot", February 2025, International Journal of Science and Research (IJSR) 14(2), DOI:10.21275/SR25207090345

[21] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, ''ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design'' 2023, arXiv:2303.07839.

[22] Md Sultanul Islam Ovi, Nafisa Anjum, Tasmina Haque Bithe, Md. Mahabubur Rahman, Mst. Shahnaj Akter Smrity in 'Benchmarking ChatGPT, Codeium, and GitHub Copilot: A Comparative Study of AI Driven Programming and Debugging Assistants', arXiv:2409.19922, 2024

[23] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, et al., "Language models are few-shot learners", Proc. NIPS, pp. 1877-1901, 2020.

[24] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever (2018). Improving language understanding by generative pre-training, https://cdn.openai.com/research-covers/

[25] Radford A., Wu J., Child R., Luan D., Amodei D. et al. (2019). Language models are unsupervised multitask learners. OpenAI. https://cdn.openai.com/better-language-models/

[26] 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.

[27] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021

[28] Jingyao Li, Pengguang Chen, Bin Xia, Hong Xu, Jiaya Jia et al. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. arXiv preprint arXiv:2312.15960, 2023.

[29] Nijkamp E., Pang B., Hayashi H., Tu L., Wang, H. et al. (2022). CodeGen: An open large language model for code with multi-turn program synthesis. https://arxiv.org/abs/2203.13474

[30] Wei Y., Wang Z., Liu J., Ding Y. & Zhang, L. (2023). Magicoder: Empowering Code Generation with OSS-INSTRUCT. ArXiv, abs/2312. 02120.

[31] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid et al. (2022). Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. arXiv preprint arXiv:2206.04615.

[32] Touvron H., Lavril T., Izacard G., Martinet X., Lachaux et al. (2023). LLaMA: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.

[33] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter et al. (2022). Chain-of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903.

[34] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang et al. (2023). A survey of large language models. arXiv preprint arXiv:2303.18223.

[35] K. S. Kalyan "A survey of GPT-3 family large language models including ChatGPT and GPT-4," arXiv preprint arXiv:2310.12321, Oct. 2023. [Online]. Available: https://arxiv.org/abs/2310.12321

[36] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng et al. (2023). Efficient large language models: A survey. arXiv preprint arXiv:2312.03863.

[37] Ipek Ozkaya, Anita Carleton, John E. Robert, and Douglas Schmidt (2023). Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?. Carnegie Mellon University, Software Engineering Institute's Insights. Retrieved from https://doi.org/10.58012/6n1p-pw64.

[38] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang et al. (2024). Large Language Models for Software Engineering: A Systematic Literature Review. arXiv preprint arXiv:2308.10620.

[39] Nguyen Van Viet, Vinh The Nguyen (2024). Large language models in software engineering: A systematic review and vision. Journal of Education For Sustainable Innovation 2(2):146-156 DOI:10.56916/jesi.v2i2.968

[40] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, John Grundy (2020). Explainable AI for Software Engineering. arXiv preprint arXiv:2012.01614.

[41] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng et al. (2023). A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv preprint arXiv:2311.10372.

[42] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, Earl T. Barr et al. (2024). Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 220–232).

[43] Lenz Belzner, Thomas Gabor, Martin Wirsing (2024). Large Language Model Assisted Software Engineering October 2023, License CC BY-NC-ND 4.0

[44] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright et al. "Training language models to follow instructions with human feedback," in NeurIPS, New Orleans, USA, 2022, pp. 27 730–27 744.

[45] Tianlin Li, Qian Liu, Tianyu Pang, Chao Du, Qing Guo et al. "Purifying large language models by ensembling a small language model," 2024, arXiv:2402.14845.

[46] Michael R. Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, Patanamon Thongtanunam : "Automatic Programming: Large Language Models and Beyond" in ACM Journals, Accepted on 10 October 2024, https://doi.org/10.1145/3708519

[47] Frank F. Xu, Uri Alon, Graham Neubig, Vincent Josua Hellendoorn "A systematic evaluation of large language models of code" in MAPS 2022: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming Pages 1 - 10 https://doi.org/10.1145/3520312.3534862

[48] Juan Cruz-Benito, Sanjay Vishwakarma ,Francisco Martin-Fernandez and Ismael Faro in "Automated Source Code Generation and Auto-Completion Using Deep Learning: Comparing and Discussing Current Language Model-Related Approaches", AI 2021, 2(1), 1-16; https://doi.org/10.3390/ai2010001

[49] Samuel Holt, Max Ruiz Luyten, Mihaela van der Schaar in "L2MAC: Large Language Model Automatic Computer for Extensive Code Generation".arXiv:2310.02003

[50] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang et al. in "Self-Planning Code Generation with Large Language Models" in ACM Transactions on Software Engineering and Methodology, Volume 33, Issue 7 Article No.: 182, Pages 1 - 30 https://doi.org/10.1145/3672456

[51] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, Lingming Zhang in "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation" part of Advances in Neural Information Processing Systems 36 (NeurIPS 2023)

[52] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, Sunghun Kim (2024). A Survey on Large Language Models for Code Generation. arXiv preprint arXiv: 2406. 00515

[53] Mari Ashiga, Wei Jie, Fan Wu, Vardan Voskanyan, Fateme Dinmohammadi et al. (2025). Ensemble Learning for Large Language Models in Text and Code Generation: A Survey. arXiv preprint arXiv:2503.13505.