A Systematic Survey on Large Language Models for Code Generation

Sardar K. Jabrw and Qusay I. Sarhan†

Department of Computer Science, College of Science, University of Duhok, Duhok, Kurdistan Region – F.R. Iraq

Abstract—**The rapid development of large language models (LLMs) has transformed code generation, offering powerful tools for automating software development tasks. However, evaluating generated code's quality, security, and effectiveness remains a significant challenge. The present systematic survey comprehensively analyses studies published between 2021 and 2024, focusing on utilizing LLMs in the code generation process. The survey explored ten research questions, such as the most commonly used programming languages, the metrics employed to evaluate the quality of code, and scenarios in which LLMs are applied by developers during the software development process, outlining the scope in which prompt engineering influences code generation and security concerns with the types of benchmarks, models evaluated, and code analysis tools used in studies. The findings indicate that the most frequently used evaluation metrics in code generation are Pass@k and Bilingual Evaluation Understudy. It also shows that Python, Java, and C++ are the most widely used languages. Furthermore, identifying security vulnerabilities and establishing robust evaluation metrics remain challenges. This survey underlines present practices, detects gaps, and suggests future research to enhance the reliability and security of code generated by LLMs in real-world applications.**

Index Terms **– Benchmarking, Code Generation, Evaluation Metrics, Large Language Models.**

Introduction

The advent of large language models (LLMs) has revolutionized the field of code generation, offering unprecedented capabilities in automating software development tasks (Brown et al., 2020). These models have proven to be remarkable in generating code from natural language descriptions, completing code snippets, and even repairing errors in code (Chen et al., 2021). This enables software developers to focus on complex tasks in their code (Mendes, Souza and De Souza, 2024). On the other hand, evaluating the efficiency, quality, and security of the generated code is still challenging (Clark et al., 2024). Some of the main problems that researchers and developers are dealing with are the absence of standardized evaluation metrics and the difficulty in ensuring functional correctness, security, and maintainability of the generated code (Paul, Zhu and Bayley, 2024a).

This systematic survey aims to review the literature published between 2021 and 2024, focusing on using LLMs for code generation. With ten specific research questions (RQs), which examine the most frequently used programming languages in evaluations, the metrics used for evaluating code quality, and the scenarios in which developers apply LLMs during the software development process. Furthermore, the survey explores the impact of different prompts on the code generation process, the security of the generated code, the characteristics of the benchmarks used for evaluations, the LLMs, which had been evaluated, and the code analysis tools used in studies.

This survey uses a methodology that involves a comprehensive literature survey, conducted using well-known literature databases, to identify, select, and analyze relevant publications. The findings will contribute to a deeper understanding of the capabilities and limitations of LLMs in code generation, highlighting areas for improvement and suggesting future research directions. Moreover, it aims to serve as a valuable resource for researchers and developers by providing insights regarding the present use of LLMs for code generation and guiding future efforts to improve the reliability and security of the generated code.

The survey is motivated by multiple factors, as follows:

a) The growing application of LLMs in generating, completing, and optimizing source code necessitates a detailed analysis of evaluation methodologies to assess their functional correctness, security, and maintainability.
b) A comprehensive survey study can assist researchers and developers by consolidating knowledge on evaluation approaches, highlighting strengths, limitations, and opportunities for improving the field.
c) Despite the rapid progress of LLMs, there is still a noticeable gap in survey studies specifically addressing the evaluation criteria, benchmarks, and metrics for code generation, making this study a significant contribution to advancing the field.

The remaining parts of this survey paper are organized as follows: Section II defines the relevant research for this study. Section III describes the research methodology employed in this study. Section IV defines the findings and outcomes of

ARO-The Scientific Journal of Koya University
Vol. XIII, No. 2 (2025), Article ID: ARO.12159. 17 pages
DOI: 10.14500/aro.12159
Received 29 March 2025; Accepted 22 July 2025
Regular review paper; Published: 06 August 2025
†Corresponding author's e-mail: qusay.sarhan@uod.ac

the study. Section V addresses the risks related to validity. The study's conclusions are presented in Section VI.

II. Related Works

This section briefly reviews the relevant research on this topic. Foundational surveys, such as (Fan et al., 2023) and (Chowdhury and Haque, 2023), provided an overview of LLM applications across domains, including code synthesis. These studies highlight the benefits and limitations of using LLMs for code generation. In their studies, (Nazir and Wang, 2023) and (Kalyan, 2024) further expanded on this by discussing the development and capabilities of ChatGPT and the GPT-3 family, exploring their success across multiple fields, such as education, healthcare, and legal reasoning. They also addressed ChatGPT's limitations, such as the generation of false information, biases, and other ethical issues. Similarly, (Wang and Chen, 2023) explored the use of LLMs in code generation and highlighted three main applications: Generating code from natural language descriptions, completing code snippets, and automatically repairing bugs.

Despite impressive advancements, several studies reveal core challenges. A prominent concern is evaluation: How to reliably assess the quality, correctness, and efficiency of generated code. In their study (Paul, Zhu and Bayley, 2024a) specifically emphasized the growing reliance on LLMs in automated software engineering tasks, also focusing on the limitations of existing evaluation metrics. Similarly, (Lu et al., 2024) thoroughly examined the datasets used to evaluate LLMs for code generation, classifying datasets that reflect diverse programming skills and emphasizing the mismatch between present benchmarks and real-world scenarios. Similarly, (Chang et al., 2024) detailed the application of LLM evaluation techniques, focusing on three dimensions of what, where, and how to evaluate them. They have collected and summarized tasks in various areas, such as natural language problems, reasoning, medical usage, ethics, education, and even agent applications. Ethical and security concerns also arise. Another study by (Yao et al., 2024) provided a comprehensive review of LLMs, categorizing their impact into "The Good" (beneficial applications in security), "The Bad" (offensive uses), and "The Ugly" (vulnerabilities and defenses). They highlighted that LLMs enhance code security and data privacy, often outperforming traditional methods, but are also exploited for attacks, especially user-level ones, due to their human-like reasoning.

Recent surveys have aimed to systematize knowledge across multiple subdomains. In their study (López Espejel et al., 2023) provided a thorough review of state-of-the-art methods for generating Java code from natural language text. The methods are divided into two major groups: Recurrent Neural Network (RNN)-based and transformer-based. The Transformer-based methods are divided into encoder-only, decoder-only, and encoder-decoder models. The review traces the progress made in using deep learning models for Java code generation, concentrating on method development, advantages, and disadvantages. Similarly, (Wan et al., 2024) covered a more detailed review of deep learning in code intelligence, especially regarding code completion, code representation learning, code search, code summarization, type inference, program synthesis, deep learning tasks, etc. It depicts the development of neural architectures from RNNs and convolutional neural networks to modern Transformers and graph neural networks. In their study, (Sharma et al., 2024) through a systematic review, they categorized twelve software engineering tasks, including code completion, program synthesis, and vulnerability analysis. Their findings indicate growing hopes for using machine learning techniques for source code analysis, and they pointed out factors, such as standard dataset existence, reproducibility, and hardware resources as challenges. The authors in (Hou et al., 2024) offer a systematic literature review of 395 papers on LLMs in software engineering, identifying 85 distinct tasks and affirming the dominance of decoder-only models, such as GPT-4 in development and repair tasks.

Our systematic survey distinguishes itself from prior studies by delivering a comprehensive research analysis that utilizes LLMs in the code generation process. While existing studies often focus on isolated aspects – such as evaluation metrics (Paul, Zhu and Bayley, 2024a; Wang and Chen, 2023), applications (Fan et al., 2023), or datasets (Lu et al., 2024); this survey systematically addresses 10 interconnected RQs. Unlike surveys limited to pre-2023 research (Fan et al., 2023; Chowdhury and Haque, 2023), the present survey analysis incorporates recent developments by covering studies published between 2021 and the end of 2024, capturing significant progress after the release of models, such as GPT-4 and CodeLlama. In addition, it uniquely identifies code analysis tools used (e.g., CodeQL) in studies and their limitations, categorizes and analyzes evaluation metrics, explores prompt strategies along with their benefits and limitations, and investigates the security of generated code. Furthermore, 28 benchmarks (e.g., HumanEval, APPS) are analyzed. Finally, by highlighting key gaps and proposing future research directions, this survey aims to enhance the reliability and security of the generated code, serving as a valuable resource for researchers and developers.

III. Research Methodology

Inspired by (Petersen, Vakkalanka and Kuzniarz, 2015), the methodology used to conduct this survey includes five stages, as shown in Fig. 1. First, the initial stage involves identifying the study's objectives and various RQs. Second, the search process begins, during which a strategy is defined to identify relevant publications related to the survey topic. Third, the selection and filtering of the publications obtained in the previous stage are carried out. Next, the fourth stage is data extraction, where the relevant publications are reviewed and the key information required to answer the identified RQs is extracted. The final stage includes reporting and documenting the results. Details of these five steps, which are presented in the following subsections.

A. Identification of Research Objectives and Questions

Research objectives

This survey aims to comprehensively analyze studies published between 2021 and 2024, focusing on the LLMs for

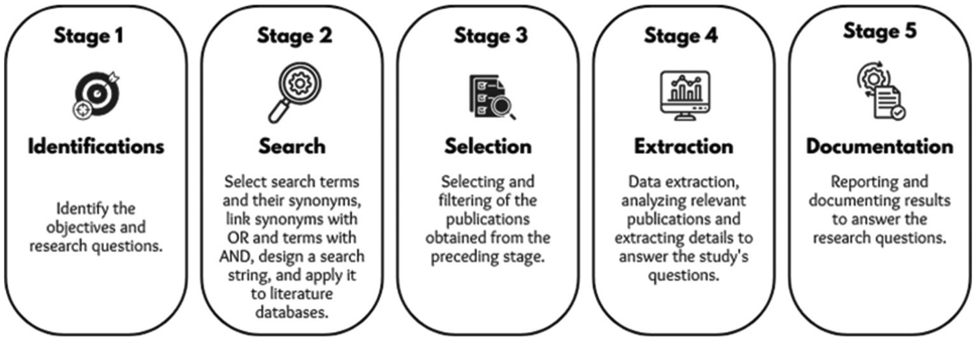
Stage 1
Identifications
Identify the objectives and research questions.
Stage 2
Search
Select search terms and their synonyms, link synonyms with OR and terms with AND, design a search string, and apply it to literature databases.
Stage 3
Selection
Selecting and filtering of the publications obtained from the preceding stage.
Stage 4
Extraction
Data extraction, analyzing relevant publications and extracting details to answer the study's questions.
Stage 5
Documentation
Reporting and documenting results to answer the research questions.

Fig. 1. Five-stage methodology for conducting the systematic survey.

code generation. The year 2021 was chosen as the starting point, as the best of our knowledge, the study (Chen et al., 2021) was the first to evaluate LLMs for code generation, which introduced Codex, a model fine-tuned on GitHub code, marking a significant advancement in the application of LLMs for code generation. By focusing on this period, the survey ensures that no significant studies are overlooked, capturing the full scope of recent developments of code generation by LLMs. By addressing ten related RQs, this survey identifies, reviews, and categorizes state-of-the-art contributions in the field of code generation, thereby contributing to the advancement of research and development in this rapidly evolving area.

RQs

This survey has identified and addressed several RQs, each of which refers to a specific facet of the topic, as outlined below:

- RQ_1: Which programming languages are used in evaluating LLMs performance for code generation tasks?
- RQ_2: Which metrics are most frequently used to evaluate the quality of LLM-generated code?
- RQ_3: For what programming scenarios, duties, and objectives are individuals using LLMs?
- RQ_4: How do different prompts impact the effectiveness of LLMs in code generation tasks?
- RQ_5: Is the code generated by LLMs secure?
- RQ_6: What are the characteristics of benchmarks used for evaluating the performance of LLMs in code generation tasks?
- RQ_7: Which LLMs are used in the evaluation of code generation?
- RQ_8: Which code analysis tools are used to evaluate code generated by LLMs?
- RQ_9: What are the challenges in evaluating LLMs for code generation?
- RQ_{10}: What are the potential future research directions for using LLMs for code generation?

B. Search Strategy

Literature sources

Well-known standard online databases, such as IEEE Xplore, Elsevier Science Direct, and ACM Digital Library, indexing publications relevant to this survey's scope, were selected as literature sources. Each database was chosen for its comprehensive collection of high-quality, peer-reviewed research in engineering, computer science, and technology, making it ideal for this survey.

Search string

The following search string was used to identify publications relevant to this survey within the literature sources:

"(Code Generation) AND (LLM OR Large Language Model OR Generative AI)"

All search terms were linked using Boolean operators. "OR" connected synonyms or related terms, while "AND" linked the main terms.

C. Paper Selection

Paper

Inclusion and exclusion criteria were defined to determine the relevance of publications. The criteria were applied based on the titles, abstracts, and full contents. Fig. 2 illustrates the quantity of included and excluded papers at each phase of the selection process. After applying the following inclusion/exclusion criteria, 74 papers were included in this survey.

Inclusion criteria

- Publications that utilize LLMs in the code generation process. These studies were selected because they focus on evaluating the performance and effectiveness of LLMs in generating code across various programming tasks.
- Publications published online from 2021 to 2024. Our literature search indicates that studies evaluating LLMs for code generation began to emerge after 2021.

Exclusion criteria

- Publications not published in English.
- Publications not directly related to the research topic.
- Publications that are not peer-reviewed (e.g., gray literature)
- Publications not published electronically.
- Duplicate publications.
- Publications without precise results.

Table I lists all the studies used in this study and the RQ/Section they support.

D. Data Extraction and Analysis

Data were systematically extracted from the selected papers and subjected to extensive analysis. This process

TABLE I
Mapping of Included Papers to Their Supported RQ/Section

#	Paper	RQ/section supported	#	Paper	RQ/section supported
1	(Afsharmazayejani et al., 2024)	RQ1, RQ7, RQ9	38	(Liu et al., 2024a)	RQ1, RQ2, RQ4, RQ6, RQ7
2	(Aggarwal et al., 2024)	RQ1, RQ2, RQ6, RQ7	39	(Liu et al., 2023)	RQ1, RQ2, RQ6, RQ7
3	(Al-Khafaji and Majeed, 2024)	RQ1, RQ7, RQ8	40	(Liu et al., 2024b)	RQ1, RQ7, RQ8, RQ9
4	(Beurer-Kellner, Vechev and Fischer, 2023)	RQ2, RQ4, RQ9	41	(López Espejel et al., 2023)	Related Work
5	(Black, Rimal and Vaidyan, 2024)	RQ1, RQ2, RQ4, RQ7	42	(Lu et al., 2024)	Related Work
6	(Bucaioni et al., 2024)	RQ1, RQ2, RQ7, RQ8	43	(MacEdo et al., 2024)	RQ1, RQ2, RQ4, RQ6, RQ7
7	(Chang et al., 2024)	Related work	44	(Majdinasab et al., 2024)	RQ1, RQ5, RQ7, RQ8
8	(Chen et al., 2021)	Related work	45	(Mendes, Souza and De Souza, 2024)	RQ3
9	(Chowdhury and Haque, 2023)	Related work	46	(Miah and Zhu, 2024)	RQ1, RQ6, RQ7
10	(Clark et al., 2024)	RQ1, RQ2, RQ6, RQ7, RQ9	47	(Moradi Dakhel et al., 2023)	RQ2, RQ6, RQ7
11	(Corso et al., 2024)	RQ1, RQ2, RQ7	48	(Moratis et al., 2024)	RQ1, RQ3, RQ6, RQ7, RQ8
12	(Cotroneo et al., 2024)	RQ1, RQ2, RQ5, RQ6, RQ7	49	(Nazir and Wang, 2023)	Related Work
13	(de-Fitero-Dominguez et al., 2024)	RQ1, RQ2, RQ7	50	(Nikolaidis et al., 2024)	RQ1, RQ2, RQ4, RQ7
14	(DeLorenzo, Gohil and Rajendran, 2024)	RQ7	51	(Niu et al., 2023)	RQ2, RQ6, RQ7
15	(Dong et al., 2024)	RQ1, RQ6, RQ7	52	(Niu et al., 2024)	RQ2, RQ4, RQ6, RQ7
16	(Du et al., 2024)	RQ1, RQ2, RQ4, RQ6, RQ7	53	(Ouyang et al., 2024)	RQ1, RQ2, RQ4, RQ6, RQ7
17	(Dumitran et al., 2024)	RQ1, RQ2, RQ7	54	(Paul, Zhu and Bayley, 2024a)	Related Work
18	(Evtikhiev et al., 2023)	RQ1, RQ2, RQ7	55	(Paul, Zhu and Bayley, 2024b)	RQ1, RQ2, RQ6, RQ7, RQ9,
19	(Fan et al., 2023)	Related work	56	(Petrovic, Konicanin and Suljovic, 2023)	RQ1, RQ7
20	(Feng et al., 2023)	RQ1, RQ7, RQ8	57	(Rai et al., 2024)	RQ1, RQ4, RQ7, RQ9
21	(Geng et al., 2023)	RQ1, RQ2	58	(Rizvi et al., 2024)	RQ1, RQ7
22	(Gu et al., 2024)	RQ1, RQ2, RQ7	59	(Sakib, Khan and Karim, 2023)	RQ1, RQ2, RQ7
23	(Guo, 2024)	RQ1, RQ2, RQ7, RQ8	60	(Sharma et al., 2024)	Related Work
24	(Hajipour et al., 2024)	RQ1, RQ5, RQ6, RQ8	61	(Siddiq et al., 2024)	RQ1, RQ3, RQ5, RQ8
25	(Hamer, D'Amorim and Williams, 2024)	RQ5, RQ7, RQ8	62	(Su et al., 2023)	RQ1, RQ2, RQ5, RQ6, RQ7, RQ8
26	(Hou et al., 2024)	Related work	63	(Tony et al., 2023)	RQ1, RQ5, RQ6, RQ7
27	(Jesse et al., 2023)	RQ4, RQ6	64	(Vijayaraghavan et al., 2024)	RQ1, RQ6, RQ7
28	(Jiang et al., 2024)	RQ1, RQ2, RQ4, RQ7	65	(Wan et al., 2024)	Related Work
29	(Jin et al., 2024)	RQ3, RQ6, RQ7	66	(Wang and Chen, 2023)	Related Work
30	(Kalyan, 2024)	Related Work	67	(Wang et al., 2024)	RQ3
31	(Kashanaki, Zakharov and Renau, 2024)	RQ1, RQ2, RQ6, RQ7	68	(Xiao et al., 2024)	RQ6
32	(Khojah et al., 2024)	RQ3, RQ4, RQ7	69	(Xu et al., 2023)	RQ1, RQ2, RQ7
33	(Khoury et al., 2023)	RQ1, RQ5, RQ7	70	(Yan, Gao and Liu, 2023)	RQ1, RQ2, RQ6, RQ7, RQ8
34	(Kou et al., 2024)	RQ1, RQ2, RQ7	71	(Yang et al., 2024)	RQ1, RQ2, RQ7
35	(Koubaa et al., 2023)	RQ1, RQ2, RQ6	72	(Yao et al., 2024)	Related Work
36	(Li et al., 2024a)	RQ1, RQ2, RQ4, RQ6, RQ7	73	(Yu et al., 2024)	RQ1, RQ2, RQ4, RQ6, RQ7
37	(Li et al., 2024b)	RQ1, RQ7	74	(Zhao et al., 2024)	RQ1, RQ6, RQ7,

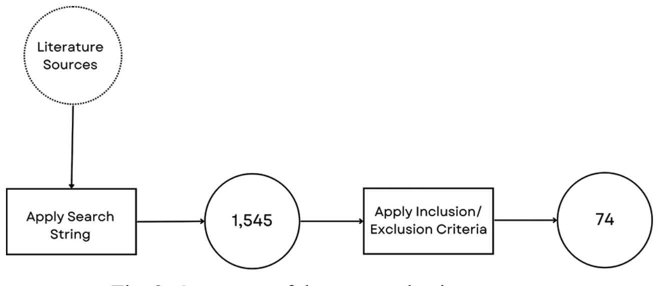

Fig. 2. Outcomes of the paper selection process.

ensured that a comprehensive and detailed understanding, enabling the study to effectively address the identified RQs with clarity and accuracy.

E. Documentation

In the final stage, the findings extracted and analyzed from the selected studies are synthesized into a structured paper that directly addresses the RQs. This includes organizing insights around key focus areas, such as programming languages, evaluation metrics, security, benchmarks, and potential future direction.

III. Result

The study addressed each identified RQ through a detailed analysis of selected publications based on the survey's findings, each RQ is summarized with a short title and discussed in its respective subsection.

A. Programming Languages for Evaluation (RQ_1)

All selected studies were reviewed to determine which programming languages were used to evaluate the LLMs' efficiency in code generation. Most studies focused on three main languages, which were 31 papers focusing on Python, 18 on Java, and 12 on C++. A few of them researched multiple programming languages, which indicated the versatility of LLMs. However, in the case of hardware programming languages, such as Verilog and very high speed integrated circuit hardware description language (VHDL), significantly fewer papers existed, with only 4 papers focusing on Verilog and 2 on VHDL. Table II shows that the programming language with the highest number of publications is Python, due to its familiarity, ease of use, and flexibility. Python is one of the most widely adopted programming languages, offering many libraries and frameworks that cover a broad range of applications. GitHub and other public repositories also offer plentiful Python code for training and testing LLMs. Moreover, well-known datasets, such as HumanEval and Mostly Basic Python Problems (MBPP) primarily use Python because it is the most dominant coding language. Its widespread use in education and research ensures that researchers are familiar with the language, reinforcing its position as the preferred choice for evaluating LLMs.

B. Metrics for Code Quality (RQ_2)

The analysis of the metrics used during evaluations reveals that a wide range of studies employed diverse metrics to assess the performance and quality of the generated code. Many of these studies employed combinations of metrics across different categories, reflecting an effort to adopt a more holistic evaluation approach. Namely, functional correctness metrics, such as pass ratio/pass@k and accuracy rate, which assess the correctness of the generated code, are among the most commonly used. Similarity metrics, such as Bilingual Evaluation Understudy (BLEU) and CodeBLEU, which evaluate the similarity of the generated codes to developer-written code, are widely used. Similarly, measuring cyclomatic complexity alongside counting lines of code (LOC) tends to be used to evaluate the logical and structural complexity of the generated code. In adition, Time and space complexity are also widely used together to assess the performance of generated code in terms of execution speed and memory usage. Moreover, generation speed and average completion time are used as indicators of LLMs' responsiveness to real-world demands and often used as usability metrics.

All reviewed studies indicate that none of the models can perform best in all types of tasks. Consequently, no single useful and successful evaluation metric has been explicitly identified. When analyzing which metrics are most robust for evaluating LLMs in code generation, it becomes clear that no single metric is universally superior; a robust evaluation necessitates a multifaceted approach.

Functional correctness metrics, such as pass ratio/pass@k are core, directly assessing if generated code passes tests, making them highly relevant for utility. However, Pass@k has limitations, including its dependence on test suite adequacy, which can be inadequate, leading to false judgments. Another significant limitation is that users do not usually run the LLM several times, so pass@k does not reflect its usability. While it demonstrates the randomness of the LLM's output, it does not align with a user's typical interactive process of generating code with an LLM, which might involve multiple attempts with input amendments until a satisfactory solution is obtained.

TABLE II
Programming Languages used in Evaluations of LLMs

#	Programming languages	Published papers	Total
1	Python	(Yan, Gao and Liu, 2023), (Nikolaidis et al., 2024), (Clark et al., 2024), (Su et al., 2023), (Xu et al., 2023), (Majdinasab et al., 2024), (Black., Rimal and Vaidyan, 2024), (Zhao et al., 2024), (Hajipour et al., 2024), (Yu et al., 2024), (Aggarwal et al., 2024), (Rai et al., 2024), (Du et al., 2024), (Al-Khafaji and Majeed, 2024), (Dumitran, Badea and Muscalu, 2024), (MacEdo et al., 2024), (Sakib, Khan and Karim, 2023), (Khoury et al., 2023), (Feng et al., 2023), (Tony et al., 2023), (Liu et al., 2024b), (Siddiq et al., 2024), (Geng et al., 2023), (Gu et al., 2024), (Ouyang et al., 2024), (Kou et al., 2024), (Dong et al., 2024), (Jiang et al., 2024), (Li et al., 2024a), (Koubaa et al., 2023), (Evtikhiev et al., 2023)	31
2	Java	(Xu et al., 2023), (Yu et al., 2024), (Rai et al., 2024), (MacEdo et al., 2024), (Corso et al., 2024), (Liu et al., 2024a), (Khoury et al., 2023), (Guo, 2024), (Liu et al., 2024b), (Siddiq et al., 2024), (Paul, Zhu and Bayley, 2024b), (Geng et al., 2023), (Gu et al., 2024), (Li et al., 2024b), (Yang et al., 2024), (Jiang et al., 2024), (Koubaa et al., 2023), (Bucaioni et al., 2024)	18
3	C++	(Rai et al., 2024), (Rizvi et al., 2024), (Dumitran, Badea and Muscalu, 2024), (MacEdo et al., 2024), (Khoury et al., 2023), (Liu et al., 2024b), (Yang et al., 2024), (Gu et al., 2024), (Li et al., 2024a), (de-Fitero-Dominguez et al., 2024), (Koubaa et al., 2023), (Bucaioni et al., 2024)	12
4	C	(Black, Rimal and Vaidyan, 2024), (Hajipour et al., 2024), (Rai et al., 2024), (MacEdo et al., 2024), (Khoury et al., 2023), (Liu et al., 2024b), (de-Fitero-Dominguez et al., 2024)	7
5	Verilog	(Afsharmazayejani et al., 2024), (Kashanaki, Zakharov and Renau, 2024), (Kashanaki, Zakharov and Renau, 2024), (Liu et al., 2023)	4
6	JavaScript	(Liu et al., 2024b), (Moratis et al., 2024), (Jiang et al., 2024)	3
7	VHDL	(Afsharmazayejani et al., 2024), (Vijayaraghavan et al., 2024)	2
8	R	(Miah and Zhu, 2024)	2
9	Arduino	(Petrovic, Konicanin and Suljovic, 2023)	1
10	Go	(MacEdo et al., 2024), (Gu et al., 2024), (Jiang et al., 2024)	3
11	Assembly Language (32)	(Cotroneo et al., 2024)	1
12	HTML	(Khoury et al., 2023)	1

Human-centric metrics, such as #attemptk (this metric focuses on the average number of user attempts to obtain a satisfactory solution), and direct human evaluation offer insights into usability, understandability, and alignment with developers' needs, yet they are time-consuming, expensive, and subjective. In addition to the Similarity metrics, such as BLEU, are often considered suboptimal for code generation that was initially designed for machine translation. BLEU primarily measures n-gram overlap between a candidate text and a reference text. While this works reasonably well for natural language, where semantic similarity often correlates with lexical overlap, code has a much stricter syntax and semantics. Hence, CodeBLEU metric attempts to address these limitations by incorporating program-specific features which extends traditional BLEU by including four sub-metrics: N-gram match (traditional BLEU), weighted n-gram match (assigning different weights to token types), abstract syntax tree match (capturing syntactic similarity), and data flow match (evaluating semantic equivalence through data flow graphs). While described as more accurate and better adapted for code generation, CodeBLEU still has recognized limitations. It can be overly strict and underestimate a model's performance. It has been found to perform no better than more generic metrics from machine translation in correlation with human assessment. N-gram-based components of CodeBLEU still suffer from a poor correlation with human scores because of their inability to capture semantic meaning. Therefore, a truly robust evaluation of LLM-generated code requires an integration of execution-based functional correctness, human judgment for qualitative aspects, domain-specific metrics, and continuous adaptation to the evolving nature of LLMs.

A truly robust evaluation of LLM-generated code requires a comprehensive and multifaceted strategy that integrates execution-based functional correctness, human judgment for qualitative aspects, domain-specific metrics (e.g., for security or creativity), and the development of dynamic and evolving evaluation systems to keep pace with the rapid advancements of LLMs and to counter issues, such as data contamination. This continuous evolution necessitates an adaptive and comprehensive evaluation strategy considering various aspects of code quality and real-world applicability.

Further details on the metrics and their categories are presented in Table III, and they are briefly discussed below.

- Functional correctness: This category evaluates whether the generated code produces the correct outputs for given inputs. The most common metric in this category is Pass@k, which evaluates how many codes out of k attempts were able to pass a set of pre-defined test cases. It is popular for directly testing execution success, easy to calculate, and aligns with popular benchmarks, such as HumanEval.
- Syntactic closeness/similarity: This assesses how structurally a generated code is similar to a reference code (developer-written code) in terms of syntax, variable names, code length, etc. The dominant metric for measuring code similarity is BLEU, which was discussed above.
- Code complexity: This quantifies how complex code is to read, debug, or maintain. Lines of Code (LOC) is the most used metric due to its simplicity, offering a universally accessible and easy-to-calculate measure of code complexity. However, it does not account for code quality, structure, or readability, and more extended code is not necessarily more complex or challenging to maintain. In addition, LOC can be influenced by formatting styles or language syntax, making it a blunt tool for deeper evaluation of code efficiency.
- Code performance: This measures the computational efficiency (e.g., speed and memory usage) of the generated code. Time complexity (e.g., runtime scaling) is mostly adopted, as slow code can significantly undermine a solution's practicality. It reflects algorithmic quality and is

TABLE III
Metrics used in LLM Evaluations and Their Categories

#	Categories	Metrics	Published Papers
1	Functional Correctness	Pass@k	(Zhao et al., 2024), (Yu et al., 2024), (Aggarwal et al., 2024), (Du et al., 2024), (Niu et al., 2024), (Paul, Zhu and Bayley, 2024b), (Miah and Zhu, 2024), (Liu et al., 2023), (Dong et al., 2024), (Jiang et al., 2024), (Li et al., 2024a)
		Accuracy rate	(Yan, Gao and Liu, 2023), (Black, Rimal and Vaidyan, 2024), (Niu et al., 2023)
		Acc@K	(Yu et al., 2024)
		Pass@TopK	(Moradi Dakhel et al., 2023)
		Success rate/pass ratio	(Sakib, Khan and Karim, 2023), (Ouyang et al., 2024), (Jiang et al., 2024), (Koubaa et al., 2023), (Bucaioni et al., 2024), (Moradi Dakhel et al., 2023)
		Exact match accuracy	(Yang et al., 2024), (Cotroneo et al., 2024)
		Computational accuracy	(Yang et al., 2024), (MacEdo et al., 2024)
		Compilation accuracy	(Cotroneo et al., 2024)
		Compilation rate, match success rate, code extraction success rate	(MacEdo et al., 2024)
2	Syntactic Closeness/ Similarity	BLUE	(Yu et al., 2024), (Liu et al., 2024a), (Niu et al., 2023), (Geng et al., 2023), (Gu et al., 2024), (Liu et al., 2023), (Gu et al., 2024), (Cotroneo et al., 2024), (Evtikhiev et al., 2023)
		CodeBLEU	(Al-Khafaji and Majeed, 2024), (Corso et al., 2024), (Liu et al., 2024a), (Gu et al., 2024), (Jiang et al., 2024), (Evtikhiev et al., 2023)
		text2vec	(Yan, Gao and Liu, 2023)
		Jaccard similarity	(Yu et al., 2024)
		SacreBLEU	(Cotroneo et al., 2024)
		Normalized Levenshtein similarity	(Corso et al., 2024), (Ouyang et al., 2024)
		OpenAI Text-Embedding Ada-002	(Xu et al., 2023)
		ROUGE	(Geng et al., 2023), (Evtikhiev et al., 2023)
		Perfect predictions (PP)	(de-Fitero-Dominguez et al., 2024)
		METEOR	(Geng et al., 2023), (Evtikhiev et al., 2023)
		RUBY	(Evtikhiev et al., 2023)
		ChrF	(Evtikhiev et al., 2023)
		Longest common subsequence (LCS)	(Ouyang et al., 2024)
		San Martino's token overlapping metrics	(Kou et al., 2024)
		Abstract syntax trees (AST)	(Moradi Dakhel et al., 2023)
		SentenceBERT+ cosine similarity (SBCS)	(Gu et al., 2024)

(*Contd...*)

TABLE III
(*Continued*)

#	Categories	Metrics	Published Papers
3	Code Complexity	Lines of code	(Nikolaidis et al., 2024), (Dumitran, Badea and Muscalu, 2024), (Corso et al., 2024), (Paul, Zhu and Bayley, 2024b), (Kashanaki, Zakharov and Renau, 2024), (Beurer-Kellner, Vechev and Fischer, 2023)
		Cyclomatic complexity	(Nikolaidis et al., 2024), (Su et al., 2023), (Paul, Zhu and Bayley, 2024b), (Moradi Dakhel et al., 2023)
		Token count	(Nikolaidis et al., 2024), (Xu et al., 2023)
		Cognitive complexity	(Su et al., 2023), (Paul, Zhu and Bayley, 2024b)
		Line width	(Dumitran, Badea and Muscalu, 2024)
		Halstead complexity metrics	(Clark et al., 2024)
		McCabe cyclomatic complexity	(Corso et al., 2024)
		Words count	(Xu et al., 2023)
4	Code Performance	Time complexity	(Nikolaidis et al., 2024), (Sakib, Khan and Karim, 2023), (Corso et al., 2024), (Guo, 2024), (Niu et al., 2024), (Bucaioni et al., 2024)
		Space complexity	(Nikolaidis et al., 2024), (Sakib, Khan and Karim, 2023), (Guo, 2024), (Bucaioni et al., 2024)
5	Usability and Productivity	Generation speed	(Black, Rimal and Vaidyan, 2024), (Zhao et al., 2024), (Al-Khafaji and Majeed, 2024), (Miah and Zhu, 2024)
		Average completion time	(Miah and Zhu, 2024)
		Response received	(Moradi Dakhel et al., 2023)
		#AttemptK	(Miah and Zhu, 2024), (Guo, 2024)

easier to analyze than memory usage, particularly in dynamic environments.

- Usability and productivity: This evaluates how well LLMs can aid developers in their coding tasks. Generation speed (how fast a code is generated) is a key, as faster outputs boost productivity in real time. It is tangible and critical for adoption, unlike subjective usability metrics.

C. Use Cases and Applications (RQ_3)

This RQ aims to identify the use cases of LLMs in code generation and to understand how developers apply these models in practice, based on an analysis of selected studies. In various studies (Siddiq et al., 2024), (Jin et al., 2024), and (Moratis et al., 2024), the authors used the DevGPT dataset, constructed from real developers' conversations with ChatGPT, to identify developer use cases. Furthermore, (Khojah et al., 2024), (Wang et al., 2024), and (Mendes, Souza and De Souza, 2024) conducted studies involving real developers to pose questions and carried out surveys to evaluate the effectiveness of ChatGPT in supporting software development tasks and understanding its usage patterns. The following are the most common use cases, and they are summarized in Table IV:

TABLE IV
Summary of LLM Usage Categories and Actions

#	Category	Action
1	Code generation and refactoring	Write this code Improve this code Fix this issue Solve the following problem Help me fix it
2	Learning, explanation, and educational support	Example usage (API or Objects) Explain this code Ask questions to find the correct way
3	Code optimization, formatting, and quality assurance	Request improvements Request more description Add specific instructions Request verification Point mistake then request a fix Test my input
4	Documentation and deployment	Add more context Request examples
5	Specialized tasks and miscellaneous use cases	Request another generation Networking, bioinformatics, or APIs.

- Code generation and refactoring: Survey findings indicate that developers extensively utilize ChatGPT for generating and refactoring code. For example, developers rely on ChatGPT to create entire functions or code snippets integrated into their projects. LLMs also improve existing code by enhancing its efficiency, readability, and performance. For example, ChatGPT's suggestions help developers write more efficient code (Jin et al., 2024), and often replace web searches, speeding up access to relevant answers (Wang et al., 2024).
- Learning, explanation, and educational support: Studies reveal that LLMs are valuable educational resources for developers aiming to learn new programming libraries, frameworks, or application programming interfaces (APIs). For example, by using ChatGPT, developers gain insights into the usage of functions and frameworks, including machine learning libraries (e.g., PyTorch and TensorFlow) (Siddiq et al., 2024). ChatGPT explains different portions of code in detail, outlining how and why specific techniques are used (Moratis et al., 2024).
- Code optimization, formatting, and quality assurance: One of ChatGPT's primary strengths is assisting with code optimization and formatting. Developers use it to improve code structure, ensuring adherence to best practices and alignment with coding standards. ChatGPT facilitates the reorganization, corrects indentation, and applies consistent naming conventions (Siddiq et al., 2024). It also contributes to code reviews, offering recommendations to improve code quality, performance, and readability.
- Documentation and deployment: Studies highlight LLM's role in documentation and deployment support throughout the software development lifecycle. Developers use ChatGPT to write and modify documentation, such as README files and code comments, simplifying the communication of complex concepts and facilitating code sharing (Jin et al., 2024). Besides,
- Specialized tasks and miscellaneous use cases: Survey findings indicate that ChatGPT is used for tasks beyond code

writing, including networking, data manipulation, and data streaming operations. Developers also rely on ChatGPT to tackle complex Software Development Kit (SDK) and API challenges, such as those involving the Amazon Web Services (AWS) SDK (Boto3) and Nylas SDK (Siddiq et al., 2024).

D. Prompt Design and Effectiveness (RQ_4)

Many studies have thoroughly examined the impact of different prompts on the effectiveness of LLMs in code generation tasks. The study by (Black, Rimal and Vaidyan, 2024) highlights the role of security-focused prompts in improving the security of generated code, demonstrating that prompts explicitly addressing security flaws significantly reduce vulnerabilities. Similarly, (Yu et al., 2024) explore how the choice between original and human-labeled docstrings affects LLMs' performance, revealing that models trained on single-language corpora perform better when prompted with semantically similar instructions. In their study, (Liu et al., 2024a) evaluate three levels of prompts for guiding ChatGPT, showing that carefully constructed prompts can improve code generation performance, particularly in Text-to-Code (T2C) tasks. Similarly, (Jesse et al., 2023) compare traditional prompting techniques with chain-of-thought prompts, demonstrating that optimization-focused prompts improve runtime efficiency, especially for complex coding problems. The authors in (Li et al., 2024a) propose a novel prompting technique called structured chain-of-thought prompting, which improves the performance of LLMs in code generation tasks.

However, even the most carefully engineered prompts can sometimes fail under certain conditions. When the task description is under-specified, the model may unpredictably fill in missing details and produce code that diverges from user intent; overly long or multi-round interactions can exceed the model's token limit, causing earlier instructions to be truncated and resulting in incomplete or incorrect solutions. The non-deterministic nature of LLMs, where "the same prompt produces different answers on different inference executions," poses a significant challenge for researchers, as it makes it difficult to determine whether the proposed output is the optimal solution, especially for complex and chain-of-thought prompts. Simple template- or example-based prompts may encourage the model to mimic superficial features of the demonstrations without truly understanding the underlying logic, causing failures on edge cases. Multi-round prompts or repair loops can compound errors if the model misinterprets an earlier fix, leading to loops of incomplete or contradictory edits. A summary of each primary strategy, its benefits, and its common pitfalls

TABLE V
Prompts, Strategies, Benefits, and Common Limitations

#	Prompting category	Strategy	Typical benefits	Common limitations/failure modes
1	Core prompting strategies	Zero-Shot prompting (Ouyang et al., 2024)	It provides a quick baseline output with no examples and is ideal for rapid prototyping and broad coverage.	Poor accuracy on complex tasks; highly non-deterministic.
		Few-Shot prompting (Beurer-Kellner, Vechev and Fischer, 2023)	Leverages a handful of examples to boost relevance, style consistency, and alignment with the desired format.	Tends to overfit to the provided examples and is highly sensitive to example choice and ordering.
		Chain-of-Thought (CoT) prompting (Jiang et al., 2024)	Breaks down complex tasks into step-by-step reasoning, improving correctness on multi-step logic.	Can produce lengthy, context-heavy rationales that waste the model's token.
		Structured CoT (SCoT) prompting (Li et al., 2024a)	Embeds explicit programming structures (loops, branches) into the reasoning, yielding clearer, accurate code.	Requires significant effort to design effective structures; a large prompt size can trigger truncation and loss of instructions.
		Template-based prompting (Liu et al., 2024a)	Uses fixed prompt skeletons for consistent, repeatable results on standard tasks.	Breaks down if input deviates slightly from the template, making it brittle and inflexible to evolving requirements.
2	Contextual and framing strategies	Context-rich prompting (Khojah et al., 2024)	Incorporates domain-specific details (APIs, docstrings, constraints) to enhance accuracy and relevance	Large prompts risk exceeding token limits; outdated context can mislead the model's output.
		Role-based prompting (Black, Rimal and Vaidyan, 2024)	Frames the model's persona (e.g., "As a senior engineer…") to influence tone, style, and domain focus	Limited impact on code correctness; mostly stylistic.
		Iterative/multi-round prompting (Nikolaidis et al., 2024)	Enables continuous refinement by feeding back errors, test outputs, or user corrections	Errors can compound across rounds, leading to endless correction loops.
3	Control and constraint-based strategies	Conciseness requests prompting (Liu et al., 2024a)	Produces lean, focused code snippets that are easier to review and deterministic in behavior	Reduce readability or clarity.
		Security-focused prompting (Black, Rimal and Vaidyan, 2024)	Embeds security checks and mitigations directly into the prompt, reducing vulnerability risks	It can produce overly defensive or verbose code at the expense of performance.
		Efficiency prompts (Niu et al., 2024)	Directs the model to optimize for performance (speed, memory), yielding more resource-efficient code	Sometimes conflicts with readability or maintainability.
		Format control prompting (MacEdo et al., 2024)	Enforces a consistent output format (e.g., fenced code blocks), simplifying automated parsing and evaluation	Adds prompt overhead; may limit creative solutions

is shown in Table V. The papers reviewed suggest that the success of LLMs in code generation tasks is highly dependent on prompt design. Well-constructed prompts, tailored to task complexity and model capabilities, can significantly improve accuracy, security, and optimization. It follows that the effectiveness of prompt engineering critically determines the performance of leveraging the capabilities of LLMs.

E. Security of LLM-Generated Code (RQ_5)

Several studies (Su et al., 2023), (Majdinasab et al., 2024), (Hajipour et al., 2024), (Khoury et al., 2023), (Hamer, D'Amorim and Williams, 2024), (Tony et al., 2023), (Siddiq et al., 2024), and (Cotroneo et al., 2024) have investigated the security of generated codes, identifying various vulnerabilities, such as hard-coded credentials, improper resource management, and insecure coding patterns.

Several studies examine the security capabilities of specific LLMs. (Su et al., 2023) Evaluated codes generated by ChatGPT, Claude, Spark, and Bing AI. Their findings suggest that these newer models perform better than earlier generations regarding reliability and security. Similarly, the security of GitHub Copilot was evaluated by (Majdinasab et al., 2024), who used CodeQL and manual inspection to detect vulnerabilities in its suggestions. Their results show that despite post-processing efforts, Copilot continues to generate vulnerable code for specific categories of weaknesses. In their study (Hajipour et al., 2024) explored vulnerabilities in code generated by various LLMs, including ChatGPT, CodeGen, and Copilot, concluding that all models could produce code with exploitable flaws.

Other researchers focus on how ChatGPT performs in real-world development environments. (Siddiq et al., 2024) Analyzed developers' interactions with ChatGPT and evaluated the generated codes; they found several quality issues, such as undefined variables, insecure comments, and code that required significant revision before being used. Their study also explored the integration of this code into repositories, concluding that ChatGPT's output is typically of low quality and not merged directly due to required modifications. Similarly, (Hamer D'Amorim and Williams, 2024) compared ChatGPT-generated codes with Stack Overflow codes, concluding that while both sources pose security risks, ChatGPT demonstrates greater restraint in producing insecure patterns.

In summary, studies consistently show that LLMs are prone to generating insecure code, particularly when used without supervision. While newer models, such as ChatGPT may be less risky than community-driven platforms, such as Stack Overflow, they still cannot be trusted to produce secure code autonomously. Security tools, such as static analyzers, manual review, and robust testing remain critical. Developers must treat LLM-generated code as potentially insecure by default and adopt cautious practices when integrating such code into production systems.

F. Benchmark/Dataset Characteristics (RQ_6)

Benchmarks for evaluating LLMs in code generation are designed to assess their ability to effectively generate, understand, and execute code. These benchmarks typically have datasets containing multiple types of code snippets, functions, classes, complete programs, or even algorithms. They also provide prompts ready to be input into LLMs, often in the form of natural language or partial code, to guide the code generation process. Test cases are included to verify if the generated code is correct in terms of both syntax and semantics. In addition, reference solutions are often included as ground truth to support accurate validation and comparison. Most datasets are constructed from public repositories, coding websites, or textbooks, with links provided for reproducibility and transparency. Following a thorough review of the relevant studies, several common factors were identified across the examined benchmarks: the dataset name, type of programming language, availability of test cases, reference solutions, data sources, and overall dataset accessibility. These findings are summarized in Table VI. Among the benchmarks reviewed, HumanEval is one of the most widely used in existing studies, as it provides a standardized benchmark with well-defined Python programming problems and corresponding test cases. In addition, its structured design allows consistent comparisons across models, fostering reproducibility and comparability in research. However, newer studies avoid using HumanEval to evaluate recent models, as it may have been included in the training data of LLMs, leading to potential biases in the results.

G. Models used in Evaluations (RQ_7)

After analyzing all the relevant studies, it becomes clear which LLMs were used in each study. Models, such as ChatGPT (versions 3–5) and other specialized variants appear frequently in studies, with some research evaluating more than one LLM. Table VII highlights the LLMs used in code generation research. When selecting models for evaluation, researchers balance two axes of choice: Licensing (open-source vs. proprietary) and architecture (decoder-only vs. encoder-decoder). Proprietary models, such as OpenAI's GPT-3.5, GPT-4, Anthropic's Claude, and Google's Gemini, dominate many applied studies due to their high performance and convenient API access. GPT-4, for example, regularly tops benchmarks for functional correctness and security and generates code with extensive comments (Dumitran, Badea and Muscalu, 2024). However, the lack of transparency around their training data and internal weights makes it challenging to analyze failure modes or attention behaviors. By contrast, open-source models, such as CodeLlama, StarCoder, PolyCoder, Mistral, and CodeGen are fully inspectable and fine-tunable. Recent results show that CodeLlama-Python-7B can outperform much larger closed-weight models on benchmarks, such as HumanEval (Niu et al., 2024), and Mistral variants excel at automated vulnerability repair (de-Fitero-Dominguez et al., 2024). Yet smaller open-source models struggle to generate correct solutions from start to finish and are often constrained by their maximum context size.

Most studies focus on decoder-only transformers, which are designed to predict the next token based on a given

TABLE VI
Key Attributes of Benchmarks for Evaluating Code Generation in LLMs

#	Name	Code type	Programming language	Source of data	Test cases	Reference solutions	Dataset link
1	LLMSecEval (Tony et al., 2023)	NL prompt, code snippets	Python and C	MITRE's Top 25 Common Weakness Enumeration, Authors	-	Yes	https://github.com/tuhh-softsec/LLMSecEval/
2	DevGPT (Clark et al., 2024), (Jin et al., 2024), (Moratis et al., 2024)(Xiao et al., 2024)	Functions, classes, algorithms, and full programs	Multiple languages	Chats between developers and ChatGPT	-	-	https://github.com/NAIST-SE/DevGPT
3	OJI Dataset (Liu et al., 2023)	Full programs, functions, and algorithms.	C++, Python Code	Romanian Informatics Olympiad (2002–2023)	Yes	Yes	-
4	HumanEval (Zhao et al., 2024), (Aggarwal et al., 2024), (Niu et al., 2024) (Ouyang et al., 2024), (Dong et al., 2024), (Li et al., 2024a)	Functions	Python	Authors	Yes	Yes	https://github.com/openai/human-eval
5	Human-Eval – ET (Zhao et al., 2024)	Functions	Python	Human-Eval with additional test cases	Yes	Yes	
6	MBPP (Zhao et al., 2024), (Niu et al., 2024), (Dong et al., 2024), (Li et al., 2024a)	Functions, classes, programs	Python	Coding platforms and competitive programming problems.	Yes	Yes	http://github.com/google-research/google-research/tree/master/mbpp/
7	MBPP-ET (Zhao et al., 2024), (Dong et al., 2024)	Functions, classes, programs	Python	MBPP with additional test cases	Yes	Yes	-
8	APPS (Yan, Gao and Liu, 2023), (Ouyang et al., 2024), (Dong et al., 2024)	Functions, algorithms, and complete programs	Python	Programming platforms and online coding competitions	Yes	Yes	https://github.com/hendrycks/apps
9	LLMC Dataset (Su et al., 2023)	Functions, classes, programs	Python	LeetCode, Authors	Yes	Yes	-
10	MBPP+ (Aggarwal et al., 2024)	Functions, classes, programs	Python	Extended version of MBPP	Yes	Yes	https://github.com/evalplus/mbppplus_release
11	Bash dataset (Aggarwal et al., 2024)	Bash scripts	Bash scripts	Authors	Yes	Yes	-
12	CodeLMSec (Hajipour et al., 2024)	functions, classes, and algorithms	Python and C	Generated using GPT-4 and Code Llama-34B	-	-	https://github.com/codelmsec/codelmsec
13	CoderEval (Yu et al., 2024), (Dong et al., 2024)	functions and non-standalone functions	Python and Java	Open-source projects	Yes	Yes	https://github.com/CoderEval/CoderEval
14	ClassEval (Du et al., 2024)	Classes with multiple methods	Python	Authors	Yes	Yes	https://github.com/FudanSELab/ClassEval
15	CodeXGLUE (Niu et al., 2023)	functions, classes, and programs	Java, C#, Python, Ruby, Go, C/C++, JavaScript, PHP	open-source repositories and coding platforms	Yes	Yes	https://github.com/microsoft/CodeXGLUE
16	R Dataset (Miah and Zhu, 2024)	Functions and Algorithms	R	R programming textbooks	Yes	Yes	-
17	CodeNet (MacEdo et al., 2024)	functions, algorithms, and programs	C, C++, Go, Java, and Python	Open-source projects and coding platforms	Yes	Yes	http://github.com/IBM/Project_CodeNet/
18	T2C Dataset (Liu et al., 2024a)	functions	Java	Part of the CodeXGlue benchmark	-	Yes	https://github.com/BaoBaoGitHub/guiding-chatgpt-for-code-generation
19	C2C Dataset (Liu et al., 2024a)	functions	C# and Java	Part of the CodeXGlue benchmark	-	Yes	https://github.com/BaoBaoGitHub/guiding-chatgpt-for-code-generation
20	ManySStuBs4J (Jesse et al., 2023)	Single-statements	Java	Open-source projects	-	Yes	-
21	LeetCodeEval (Niu et al., 2024)	functions	C++	LeetCode Problems	Yes	Yes	https://github.com/NougatCA/EfficiencyEval
22	ScenEval (Paul, Zhu and Bayley, 2024b)	functions, algorithms, and programs	Java	textbooks, W3Resource, and Stack Overflow	Yes	Yes	-
23	HDLEval (Kashanaki, Zakharov and Renau, 2024)	code relevant to hardware design	Verilog, Chisel, pyRTL, and DSLX.	HDLBits	-	Yes	-

(*Contd...*)

TABLE VI
(*Continued*)

#	Name	Code type	Programming language	Source of data	Test cases	Reference solutions	Dataset link
24	VHDL-Eval (Vijayaraghavan et al., 2024)	digital logic design and hardware description tasks	VHDL	Verilog-Eval and VHDL tutorials	Yes	Yes	-
25	VerilogEval (Liu et al., 2023)	hardware design tasks	Verilog	HDLBits	Yes	Yes	https://github.com/NVlabs/verilog-eval
26	CopilotEvaluation (Moradi Dakhel et al., 2023)	Sorting algorithms, Data structures	Python	Programming courses and books	Yes	Yes	http://github.com/Copilot-Eval-Replication-Package/CopilotEvaluation/
27	Shellcode_IA32 dataset (Cotroneo et al., 2024)	Assembly code snippets	Assembly Language (IA-32)	Publicly Available Security Exploits:	-	Yes	https://github.com/dessertlab/Shellcode_IA32
28	IEEEXtreme (Koubaa et al., 2023)	functions, algorithms, and programs	Python 3, Java 7, and C++	IEEEXtreme Competition	Yes	Yes	http://www.kaggle.com/datasets/riotulab/chatgpt-evaluation-on-ieeextreme-competitions/

VHDL: Very high speed integrated circuit hardware description language

Table VII
LLMs Used in Code Generation Evaluation Studies

Model	Published papers	Total
ChatGPT – 3.5	(Yan, Gao and Liu, 2023), (Clark et al., 2024), (Black, Rimal and Vaidyan, 2024), (Aggarwal et al., 2024), (Rai et al., 2024), (Du et al., 2024), (Al-Khafaji and Majeed, 2024), (Corso et al., 2024), (Guo, 2024), (Paul, Zhu and Bayley, 2024b), (DeLorenzo, Gohil and Rajendran, 2024), (Kashanaki, Zakharov and Renau, 2024), (Li et al., 2024b), (Khojah et al., 2024), (Cotroneo et al., 2024)	14
ChatGPT 4	(Petrovic, Konicanin and Suljovic, 2023), (Zhao et al., 2024), (Du et al., 2024), (Al-Khafaji and Majeed, 2024), (Dumitran, Badea and Muscalu, 2024), (Sakib, Khan and Karim, 2023), (Niu et al., 2024), (DeLorenzo, Gohil and Rajendran, 2024), (Kashanaki, Zakharov and Renau, 2024), (Kashanaki, Zakharov and Renau, 2024), (Miah and Zhu, 2024), (Gu et al., 2024), (Li et al., 2024b), (Liu et al., 2023), (Ouyang et al., 2024), (Kou et al., 2024), (Dong et al., 2024), (Bucaioni et al., 2024)	15
ChatGPT (Version not mentioned)	(Nikolaidis et al., 2024), (Su et al., 2023), (Jin et al., 2024), (Miah and Zhu, 2024), (Feng et al., 2023), (Paul, Zhu and Bayley, 2024b), (Moratis et al., 2024), (Gu et al., 2024)	9
ChatGPT – 3.5 Turbo	(Xu et al., 2023), (Yu et al., 2024), (Liu et al., 2024a), (Khoury et al., 2023), (Hamer, D'Amorim and Williams, 2024), (Liu et al., 2024b), (Niu et al., 2024), (Kashanaki, Zakharov and Renau, 2024), (Liu et al., 2023), (Ouyang et al., 2024), (Yang et al., 2024), (Gu et al., 2024), (Dong et al., 2024), (Li et al., 2024a)	12
CodeLlama-Python-7B	(Zhao et al., 2024), (Dumitran, Badea and Muscalu, 2024), (Dumitran, Badea and Muscalu, 2024), (MacEdo et al., 2024), (Niu et al., 2024), (Vijayaraghavan et al., 2024), (DeLorenzo, Gohil and Rajendran, 2024), (Yang et al., 2024), (Gu et al., 2024), (Dong et al., 2024)	10
Codellama 34B	(Aggarwal et al., 2024), (Dumitran, Badea and Muscalu, 2024), (MacEdo et al., 2024), (Niu et al., 2024), (Vijayaraghavan et al., 2024), (Dong et al., 2024)	6
Codellama-instruct-13b	(Rizvi et al., 2024), (MacEdo et al., 2024), (Niu et al., 2024), (DeLorenzo, Gohil and Rajendran, 2024), (Yang et al., 2024)	4
Starcoder	(Aggarwal et al., 2024), (Du et al., 2024), (Dumitran, Badea and Muscalu, 2024)	3
WizardCoder 15B	(Du et al., 2024), (MacEdo et al., 2024), (Niu et al., 2024)	3
Instruct-CodeGen 16B	(Du et al., 2024), (Hamer, D'Amorim and Williams, 2024), (Kashanaki, Zakharov and Renau, 2024), (Dong et al., 2024), (Jiang et al., 2024)	5
CodeGen (350M)	(Yu et al., 2024), (Hamer, D'Amorim and Williams, 2024), (Kashanaki, Zakharov and Renau, 2024)	3
Gemini 1.0	(Black, Rimal and Vaidyan, 2024), (Al-Khafaji and Majeed, 2024), (Dumitran, Badea and Muscalu, 2024)	3
mistral-7b-instruct-2	(Rizvi et al., 2024), (Dumitran, Badea and Muscalu, 2024)	2
mixtral-8×7b-instruct	(Rizvi et al., 2024), (MacEdo et al., 2024)	2
InCoder-6B	(Zhao et al., 2024), (Du et al., 2024), (Dong et al., 2024), (Jiang et al., 2024)	4
InCoder-1.3B	(Kou et al., 2024)	
CodeParrot-1.5B	(Kou et al., 2024)	
CodeGeeX2-6B	(Zhao et al., 2024), (Hamer, D'Amorim and Williams, 2024)	2
CodeGen2.5-7B	(Zhao et al., 2024), (Vijayaraghavan et al., 2024), (Kou et al., 2024)	2
GitHub Copilot	(Majdinasab et al., 2024), (Corso et al., 2024), (Moradi Dakhel et al., 2023)	2
PolyCoder 2.7B	(Du et al., 2024), (Hamer, D'Amorim and Williams, 2024), (Kou et al., 2024), (Gu et al., 2024)	4
Codex	(Hamer, D'Amorim and Williams, 2024), (Tony et al., 2023), (Dong et al., 2024), (Evtikhiev et al., 2023)	2
ChatGPT – 3.4	(Clark et al., 2024), (Black, Rimal and Vaidyan, 2024)	2
DeepSeek-6.7B	(Zhao et al., 2024), (Dumitran, Badea and Muscalu, 2024), (Li et al., 2024a)	3
MagiCoder-6.7B	(Zhao et al., 2024), (MacEdo et al., 2024)	2
Microsoft's Bing AI	(Su et al., 2023)	1
iFLYTEC's Spark	(Su et al., 2023)	1

(*Contd...*)

TABLE VII
(*Continued*)

Model	Published papers	Total
Anthropic's Claude	(Su et al., 2023)	1
CodeT5+-Python-770M	(Zhao et al., 2024), (Niu et al., 2023), (Gu et al., 2024), (Cotroneo et al., 2024)	4
DeepSeek Coder (33B Base, 33B Instruct)	(Niu et al., 2024)	1
ChatDev	(Zhao et al., 2024)	1
PanGu-Coder (300M)	(Yu et al., 2024)	1
Codellama	(Afsharmazayejani et al., 2024), (de-Fitero-Dominguez et al., 2024)	2
Mistral 7B	(Aggarwal et al., 2024), (de-Fitero-Dominguez et al., 2024)	1
RoMistral 7B	(Dumitran, Badea and Muscalu, 2024)	2
ChatGLM 6B	(Du et al., 2024)	1
Vicuna 7B	(Du et al., 2024)	1
CodeGeeX 13B	(Du et al., 2024), (Dong et al., 2024), (Jiang et al., 2024)	3
Instruct-StarCoder 15B	(Du et al., 2024), (Li et al., 2024b), (Gu et al., 2024), (Dong et al., 2024)	4
Copilot	(Nikolaidis et al., 2024)	1
Codestral 22B	(Dumitran, Badea and Muscalu, 2024)	1
AutoCoder 6.7B	(Dumitran, Badea and Muscalu, 2024)	1
CodeQwen 1.5 7B	(Dumitran, Badea and Muscalu, 2024)	1
Yi 9B	(Dumitran, Badea and Muscalu, 2024)	1
Phi3 14B	(Dumitran, Badea and Muscalu, 2024)	1
Tabnine	(Corso et al., 2024)	1
Google Bard	(Corso et al., 2024)	1
Phi-2	(Niu et al., 2024)	1
CodeGen2.5-QLoRa	(Vijayaraghavan et al., 2024)	1
Granite-20B-Code-Instruct	(Vijayaraghavan et al., 2024)	1
Granite-20B-Code-Instruct-ICL	(Vijayaraghavan et al., 2024)	1
VeriGen (6B, 16B)	(DeLorenzo, Gohil and Rajendran, 2024)	1
TransCoder	(Yang et al., 2024)	1
AlphaCode (1.1B)	(Dong et al., 2024)(Jiang et al., 2024)	2
CodeBERT	(Cotroneo et al., 2024)	1

input context. They generate code autoregressively, token by token, based on the preceding context and generated tokens (Liu et al., 2024b). The GPTseries (GPT3/3.5/4), Codex, and open-source models, such as CodeLlama, StarCoder, and DeepSeek Coder fall into this category. These models excel at generative tasks where the output flows sequentially from a given prompt (like code completion or initial code generation from a natural language description). In contrast, encoder-decoder models (e.g., BART-derived CodeT5, PLBART, AlphaCode) comprise two main parts: An encoder and a decoder. The encoder processes the input (e.g., natural language description), converting it into a fixed-length representation (vector), and the decoder then generates the output (e.g., code) based on this representation (Liu et al., 2024a). This bidirectional encoding step can improve tasks that need a holistic understanding of the input, such as code summarization or translation between languages, but are less commonly used for pure "generate from scratch" scenarios.

H. Code Analysis Tools used in Evaluations (RQ$_8$)

The analysis identifies the code analysis tools utilized in studies. Since manually verifying generated code is time-consuming and expensive, automated tools, such as PyLint, Flake8, and SonarQube are employed for static code analysis, focusing on style, security, and maintainability. In addition, dynamic evaluation tools, such as LeetCode Online Judgment are used to evaluate the functionality and performance of generated code using pre-defined test cases. CodeQL is primarily used in many studies for its robust static analysis capabilities for identifying vulnerabilities across different programming languages, and its compatibility with various analysis scenarios makes it a reliable tool for ensuring code quality and security. Table VIII summarizes these tools and their usage in evaluating generated code.

However, these tools have limitations. PyLint, although it's precision is generally high, generates false positives and often ignores messages related to stylistic issues (e.g., whitespaces, newlines, and invalid names) and import-related messages (e.g., unused imports and missing imports) (Siddiq et al., 2024). Flake8 prioritizes quantifiable style violations (e.g., whitespace, syntax) over qualitative aspects, such as code readability (Yan, Gao and Liu, 2023). Although SonarQube supports multiple languages and continuous integration, its cognitive complexity metrics are restricted to Java, Python, and JavaScript, and it lacks suitable queries for vulnerability detection related to algorithm problems in Python and JavaScript (Liu et al., 2024b). CodeQL, while powerful for identifying security vulnerabilities and checking code quality, vulnerability capabilities that are limited to specific types, such as pointer and memory-related vulnerabilities in algorithm problems for languages, such as C, C++, and Java, and it may not have suitable queries for other languages, such as Python (Liu et al., 2024b). The LeetCode Online Judgment platform assesses runtime and memory utilization. Still, it terminates execution upon the first test failure, meaning the test case pass rates provided may serve as a

Table VIII
Studies Code Analysis Tools Used for Evaluating LLM-Generated Code

#	Tool name	Purpose	Official link
1	CodeQL (Majdinasab et al., 2024), (Hajipour et al., 2024), (Hamer, D'Amorim and Williams, 2024), (Liu et al., 2024b), (Siddiq et al., 2024)	Code vulnerability analysis	https://codeql.github.com/
2	Flake8 (Al-Khafaji and Majeed, 2024), (Feng et al., 2023)	Style guide enforcement for Python	https://flake8.pycqa.org/
3	PyLint (Siddiq et al., 2024)	Static code analysis for Python	https://pylint.pycqa.org/
4	Bandit (Siddiq et al., 2024)	Security analysis for Python	https://bandit.readthedocs.io/
5	PMD (Moratis et al., 2024)	Code quality and style checker	https://pmd.github.io/
6	CheckStyle (Guo, 2024)	Style and convention checker for Java	https://checkstyle.org/
7	LeetCode Online Judgment (Liu et al., 2024b), (Bucaioni et al., 2024)	Functional correctness evaluation	https://leetcode.com/
8	SonarQube (Su et al., 2023)	Code quality and security	https://www.sonarqube.org

lower bound rather than a complete assessment of potential correctness beyond the first failure (Liu et al., 2024b).

I. Evaluation Challenges (RQ$_9$)

Various challenges and issues arise when using LLMs for code generation, especially in evaluating their ability to generate accurate and efficient code. The main challenges are listed below:

- Lack of standardized evaluation metrics: There are no universally accepted metrics for assessing code generation. Metrics, such as Pass@k and BLEU are used in many evaluation studies, but they may not reflect the full quality, correctness, or security of the generated code (Paul, Zhu and Bayley, 2024a).
- Functional correctness versus syntactic similarity: A particular issue is the balance between functional correctness (i.e., does the code do what it was intended to do?) and syntactic similarity (i.e., how closely does the generated code resemble the reference solution?). While functional correctness is more important, it is challenging to assess automatically, particularly for complex tasks. Syntactic similarity metrics, such as BLEU may not accurately reflect the usability or correctness of the generated code, which can lead to different findings and conclusions (Paul, Zhu and Bayley, 2024b).
- Security and reliability concerns: LLMs can generate code that contains security vulnerabilities identified in the Common Weakness Enumeration (CWE). Evaluating the security of generated code requires specialized tools, such as CodeQL; still, they may produce false positives or miss specific vulnerabilities. In addition, the security of generated code can vary significantly depending on the prompt and the model's training (Majdinasab et al., 2024).
- Complexity and maintainability: Generated code may be overly complex or challenging to maintain, even if it is functionally correct. Metrics, such as cyclomatic complexity and lines of code help in understanding the complexity of the code, but they do not fully capture the readability or maintainability of the code. Human evaluation is often necessary to assess these aspects, which is costly, time-consuming, and subjective (Clark et al., 2024).
- Benchmark limitations: Many benchmarks used to evaluate LLMs are constructed from a single source or lack diversity, which may introduce bias and affect the results. Datasets, such as HumanEval and MBPP are widely used but do not cover real-world programming tasks or applications. As a result, the evaluation results are limited to certain situations. (Paul, Zhu and Bayley, 2024a).
- Prompt sensitivity: The quality of the generated code can vary significantly depending on the prompt given to the LLM. Even slight modifications to a prompt can result in significantly different outputs, making it challenging to craft inputs that consistently elicit high-quality code from LLMs. This sensitivity to prompts complicates the evaluation process, as the results may depend more on the prompt design than the model's capabilities (Liu et al., 2024a).
- Token limitations and incomplete code: LLMs have token limitations, which can lead to the generation of incomplete or cut-off code snippets. This issue is highly prevalent for more complex or larger coding tasks, as the code generated can be incomplete and fail to meet the expected output. Incomplete code is challenging to evaluate, as it may not be possible to determine its correctness or functionality (Liu et al., 2024b).
- Cross-language evaluation: LLMs are often evaluated on a single programming language (e.g., Python), but their performance may differ for other languages and application domains. Evaluating LLMs for cross-language code generation (e.g., translating code from Python to Java) introduces additional challenges, as each language has syntax, semantics, and best practices. (Rai et al., 2024).
- Evaluation of multi-round fixing: In several cases, LLMs need several prompts and rounds of fixing to get the desired output in code generation. Evaluating the effectiveness of these multi-prompt rounds is difficult since it means monitoring the progression of the output code through multiple iterations, and the final code could be influenced by the quality of the user's feedback and prompts (Liu et al., 2024b).
- Model version and updates: The specific version of the model used in evaluations can significantly impact the results. For example, newer models, such as ChatGPT may have improved capabilities, making comparing results across different versions difficult (Majdinasab et al., 2024).
- Lack of hardware code evaluation: Many LLMs are primarily evaluated on high-level programming languages (e.g., Python and Java) and lack evaluation frameworks for HDLs, such as Verilog and VHDL. This limits their applicability in hardware design and verification tasks, where specialized knowledge and syntax are required (Afsharmazayejani et al., 2024).

- Efficiency and cost: Modern LLMs face significant challenges related to efficiency and performance. Despite advances in optimizing the inference process, they still require expensive, high-performance GPUs to operate effectively (Beurer-Kellner, Vechev and Fischer, 2023). This limitation drives many researchers to rely on hosted cloud-based models, where models with advanced features are often accessible only through paid APIs, increasing computational and financial costs.

J. Future Research Directions (RQ$_{10}$)

Several possible directions for future research were identified based on the analysis of selected studies. The following points summarize these potential directions:

- Evaluation of non-determinism: Future research should address the non-deterministic (i.e., the inconsistency in the code candidates generated across different requests with identical prompts) nature of LLMs by developing evaluation frameworks that account for variability in generated output. This includes exploring methods that can reduce randomness, enhance consistency, and study the influence of prompts on non-determinism.
- Development of comprehensive metrics: Future work should develop metrics that capture not only functional correctness but also code quality, security, efficiency, readability, and maintainability. Such metrics should also be reliable and aligned with human judgment, leading to better models and more accurate performance evaluations.
- Evaluation of code summarization and documentation: Future research should explore LLMs' ability to generate accurate and useful code summaries, comments, and documentation. This includes evaluating the readability, relevance, and completeness of generated explanations.
- Development of specialized benchmarks: Future work should focus on creating benchmarks for real-world coding tasks, including complex dependencies, the use of external libraries, and even project-specific contexts.
- Evaluation of code generation for real-time systems: Future work should focus on LLMs' ability to write code for real-time systems, such as embedded and Internet of Things systems. This includes checking the power, dependability, and safety of the code output in real-time conditions.
- Multilingual code generation evaluation: Future research should compare the performance of LLMs on code generation from prompts in other languages (e.g., Arabic) and explore these models' multilingual capabilities.
- Security and correctness in code generation: Further studies may consider the impact of different prompting strategies and model selection on the security and correctness of generated code. Furthermore, examine the balance between program security and functionality.
- Quality and consistency of AI-generated code: Future research should explore the quality and consistency of ChatGPT-generated code across multiple metrics and programming languages. Further studies should investigate the real-world practicality of the code generated and how many modifications the code will need before it can be incorporated into projects.
- Evaluation of new models: New models should be evaluated for their performance in specialized tasks, such as hardware design (e.g., Verilog, VHDL), low-level programming (e.g., Assembly), and real-time systems (e.g., Arduino-based applications).
- Prompt engineering techniques: Future work should investigate manual and automated approaches to prompt design for code generation. This includes evaluating advanced techniques, such as chain-of-thought prompting, few-shot examples, and template-based methods to enhance code relevance and quality, as well as developing algorithms that automatically generate, refine, and optimize prompts based on feedback (e.g., execution success, style metrics, or user preferences).

IV. Threat to Validity

Many factors may impact the results of surveys. Thus, to prevent validity risks, the following steps were taken into consideration for this paper:

- External validity: *This is regarding the literature search*; we performed a broad literature search for this survey. Great effort was made to cover well-known sources as the primary databases to ensure good coverage and representation of relevant studies.
- Construct validity: T*his is regarding the accuracy of data extraction.* Cited sources emphasize that the precision of extracted data increases the precision of the expected survey outcomes. Human error is always possible, so it was reasonable to adopt a mixed approach. At first, data were extracted manually and then cross-checked using an LLM using manual prompts. For instance, after manually extracting which programming language is used (RQ1), the documents were prompted into a model asking, "What programming language is used in this paper?" The results were compared against manual extraction, and the differences found were addressed, if any.
- Internal validity: *This concerns study reproducibility,* a key aspect of research validity. To address this, steps include strategies for search, inclusion criteria, and data extraction processes, and the methodology used is explained in detail. This transparency will facilitate other researchers repeating the survey to authenticate and trust the findings from servicing the survey.

V. Conclusion

The evaluation of LLMs in code generation is a rapidly evolving field with significant potential to transform software development. This survey has comprehensively analyzed studies published between 2021 and 2024, addressing ten key RQs related to evaluating LLMs in code generation. The findings show that Python, Java, and C++ are the most frequently used programming languages in these evaluations, while metrics, such as Pass@k and BLEU are widely employed to assess code quality. The programming scenarios in which

LLMs were applied are diverse, including code generation, code refactoring, debugging, and even providing security support. The efficiency of LLMs relies heavily on the quality of prompt engineering, which remains a critical area for further research. Furthermore, security remains a significant concern, as generated code often contains vulnerabilities that require scrutiny and analysis using tools, such as CodeQL. While it is evident that various benchmarks are used in evaluations, the lack of standardization poses challenges for comparing results across different studies. The results achieved by ChatGPT are positive but need to be supported by more comprehensive evaluation frameworks that consider the full spectrum of code quality, security, and maintainability.

We recommend future research directions to focus on developing comprehensive metrics and benchmarks that integrate specific aspects of human cognition and real-world coding scenarios. Furthermore, addressing non-determinism alongside cross-language evaluation and multi-round code fixing is essential for the field's growth. This survey contributes to the present studies by highlighting present practices, identifying existing gaps, and proposing future research directions, aiming to improve the trustworthiness and effectiveness of the code generated by LLMs.

References

Afsharmazayejani, R., Shahmiri, M.M., Link, P., Pearce, H., and Tan, B., 2024. Toward Hardware Security Benchmarking of LLMs. In: *2024 IEEE LLM Aided Design Workshop, LAD 2024*. Institute of Electrical and Electronics Engineers Inc.

Aggarwal, P., Chatterjee, O., Dai, T., Mohapatra, P., Paulovicks, B., Blancett, B., and De Magalhaes, A., 2024. CodeSift: An LLM-Based Reference-Less Framework for Automatic Code Validation. In: *IEEE International Conference on Cloud Computing, CLOUD*. IEEE Computer Society, pp.404-410.

Al-Khafaji, N.J., and Majeed, B.K., 2024. Evaluating Large Language Models using Arabic Prompts to Generate Python Codes. In: *4th International Conference on Emerging Smart Technologies and Applications, eSmarTA 2024*. Institute of Electrical and Electronics Engineers Inc.

Beurer-Kellner, L., Vechev, M., and Fischer, M., 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7, pp. 1946-1969.

Black, G.S., Rimal, B.P., and Vaidyan, V.M., 2024. Balancing Security and Correctness in Code Generation: An Empirical Study on Commercial Large Language Models. *IEEE Transactions on Emerging Topics in Computational Intelligence*, pp.1-12.

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D., 2020. Language models are few-shot learners. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20. Curran Associates Inc., Red Hook, NY, USA.

Bucaioni, A., Ekedahl, H., Helander, V., and Nguyen, P.T., 2024. Programming with ChatGPT: How far can we go? *Machine Learning with Applications*, 15, p.100526.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Yi Chang, Zhang, Y., Yu, P.S., Yang, Q., and Xie, X., 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3), p.39.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W., 2021. *Evaluating Large Language Models Trained on Code*. Available from: https://arxiv.org/abs/2107.03374 [Last accessed on 2024 Dec 27].

Chowdhury, M.N.U.R., and Haque, A., 2023. ChatGPT: Its Applications and Limitations. In: *2023 3rd International Conference on Intelligent Technologies, CONIT 2023*. Institute of Electrical and Electronics Engineers Inc.

Clark, A., Igbokwe, D., Ross, S., and Zibran, M.F., 2024. A Quantitative Analysis of Quality and Consistency in AI-generated Code. In: *Proceedings - 2024 7th International Conference on Software and System Engineering, ICoSSE 2024*. Institute of Electrical and Electronics Engineers Inc., pp.37-41.

Corso, V., Mariani, L., Micucci, D., and Riganelli, O., 2024. Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants. In: *IEEE International Conference on Program Comprehension*. IEEE Computer Society, pp.13-23.

Cotroneo, D., Foggia, A., Improta, C., Liguori, P., and Natella, R., 2024. Automating the correctness assessment of AI-generated code for security contexts. *Journal of Systems and Software*, 216, p.112113.

De-Fitero-Dominguez, D., Garcia-Lopez, E., Garcia-Cabot, A., and Martinez-Herraiz, J.J., 2024. Enhanced automated code vulnerability repair using large language models. *Engineering Applications of Artificial Intelligence*, 138, p.109291.

DeLorenzo, M., Gohil, V., and Rajendran, J., 2024. CreativEval: Evaluating creativity of LLM-based hardware code generation. Proceedings of the 2024 IEEE LLM Aided Design Workshop (LAD), San Jose, CA, USA, pp.1-5.

Dong, Y., Jiang, X., Jin, Z., and Li, G., 2024. Self-collaboration Code Generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 33, p.189.

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., and Lou, Y., 2024. Evaluating Large Language Models in Class-Level Code Generation. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp.982-994.

Dumitran, A.M., Badea, A.C., and Muscalu, S.G., 2024. Evaluating the Performance of Large Language Models in Competitive Programming: A Multi-Year, Multi-Grade Analysis. In: *18th International Conference on INnovations in Intelligent SysTems and Applications, INISTA 2024*. Institute of Electrical and Electronics Engineers Inc.

Evtikhiev, M., Bogomolov, E., Sokolov, Y., and Bryksin, T., 2023. Out of the BLEU: How should we assess quality of the code generation models? *Journal of Systems and Software*, 203, p.111741.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J.M., 2023. Large Language Models for Software Engineering: Survey and Open Problems. In: *Proceedings - 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023*. Institute of Electrical and Electronics Engineers Inc., pp.31-53.

Feng, Y., Vanam, S., Cherukupally, M., Zheng, W., Qiu, M., and Chen, H., 2023. Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. In: *Proceedings - International Computer Software and Applications Conference*. IEEE Computer Society, pp.876-885.

Geng, M., Wang, S., Dong, D., Wang, H., Cao, S., Zhang, K., and Jin, Z., 2023. Interpretation-based Code Summarization. In: *IEEE International Conference on Program Comprehension*. IEEE Computer Society, pp.113-124.

Gu, X., Chen, M., Lin, Y., Hu, Y., Zhang, H., Wan, C., Wei, Z., Xu, Y., and Wang, J., 2024. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and*

Methodology, 34, p.78.

Guo, M., 2024. Java Web Programming with ChatGPT. In: *2024 5th International Conference on Mechatronics Technology and Intelligent Manufacturing, ICMTIM 2024*. Institute of Electrical and Electronics Engineers Inc., pp.834-838.

Hajipour, H., Hassler, K., Holz, T., Schonherr, L., and Fritz, M., 2024. CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. In: *Proceedings - IEEE Conference on Safe and Trustworthy Machine Learning, SaTML 2024*. Institute of Electrical and Electronics Engineers Inc., pp.684-709.

Hamer, S., D'Amorim, M., and Williams, L., 2024. Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. In: *Proceedings - 45th IEEE Symposium on Security and Privacy Workshops, SPW 2024*. Institute of Electrical and Electronics Engineers Inc., pp.87-94.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H., 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), p.1-79.

Jesse, K., Ahmed, T., Devanbu, P.T., and Morgan, E., 2023. Large Language Models and Simple, Stupid Bugs. In: *Proceedings - 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR 2023*. Institute of Electrical and Electronics Engineers Inc., pp.563-575.

Jiang, X., Dong, Y., Wang, L., Zheng, F., Shang, Q., Li, G., Jin, Z., and Jiao, W., 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33, p.182.

Jin, K., Wang, C.Y., Pham, H.V., and Hemmati, H., 2024. Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation. In: *Proceedings - 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR 2024*. Institute of Electrical and Electronics Engineers Inc., pp.167-171.

Kalyan, K.S., 2024. A survey of GPT-3 family large language models including ChatGPT and GPT-4. *Natural Language Processing Journal*, 6, p.100048.

Kashanaki, F.R., Zakharov, M., and Renau, J., 2024. HDLEval Benchmarking LLMs for Multiple HDLs. In: *2024 IEEE LLM Aided Design Workshop, LAD 2024*. Institute of Electrical and Electronics Engineers Inc.

Khojah, R., Mohamad, M., Leitner, P., and De Oliveira Neto, F.G., 2024. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. *Proceedings of the ACM on Software Engineering*, 1(FSE), pp.1819-1840.

Khoury, R., Avila, A.R., Brunelle, J., and Camara, B.M., 2023. How Secure is Code Generated by ChatGPT? In: *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*. Institute of Electrical and Electronics Engineers Inc., pp.2445-2451.

Kou, B., Chen, S., Wang, Z., Ma, L., and Zhang, T., 2024. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering*, 1, pp.2261-2284.

Koubaa, A., Qureshi, B., Ammar, A., Khan, Z., Boulila, W., and Ghouti, L., 2023. Humans are still better than ChatGPT: Case of the IEEEXtreme competition. *Heliyon*, 9(11), p.e21624.

Li, J., Li, G., Li, Y., and Jin, Z., 2024a. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34, p.34.

Li, J., Zhang, Y., Karas, Z., Mcmillan, C., Leach, K., and Huang, Y., 2024b. Do Machines and Humans Focus on Similar Code? Exploring Explainability of Large Language Models in Code Summarization. In: *IEEE International Conference on Program Comprehension*. IEEE Computer Society, pp.47-51.

Liu, C., Bao, X., Zhang, H., Zhang, N., Hu, H., Zhang, X., and Yan, M., 2024a. Guiding ChatGPT for Better Code Generation: An Empirical Study. In: *Proceedings - 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024*. Institute of Electrical and Electronics Engineers Inc., pp.102-113.

Liu, M., Pinckney, N., Khailany, B., and Ren, H., 2023. Invited Paper: VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*. Institute of Electrical and Electronics Engineers Inc.

Liu, Z., Tang, Y., Luo, X., Zhou, Y., and Zhang, L.F., 2024b. No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. *IEEE Transactions on Software Engineering*, 50(6), pp.1548-1584.

López Espejel, J., Yahaya Alassan, M.S., Chouham, E.M., Dahhane, W., and Ettifouri, E.H., 2023. A comprehensive review of state-of-the-art methods for Java code generation from natural language text. *Natural Language Processing Journal*, 3, p.100013.

Lu, Y., Sun, C., Yan, Y., Zhu, H., Song, D., Peng, Q., Yu, L., Wang, X., Jiang, J., and Ye, X., 2024. A Comprehensive Survey of Datasets for Large Language Model Evaluation. In: *2024 5th Information Communication Technologies Conference, ICTC 2024*. Institute of Electrical and Electronics Engineers Inc., pp.330-336.

MacEdo, M., Tian, Y., Cogo, F., and Adams, B., 2024. Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation. In: *Proceedings - 2024 IEEE/ACM 1st International Conference on AI Foundation Models and Software Engineering, FORGE 2024*. Association for Computing Machinery, Inc., pp.57-68.

Majdinasab, V., Bishop, M.J., Rasheed, S., Moradidakhel, A., Tahir, A., and Khomh, F., 2024. Assessing the Security of GitHub Copilot's Generated Code-A Targeted Replication Study. In: *Proceedings - 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024*. Institute of Electrical and Electronics Engineers Inc., pp.435-444.

Mendes, W., Souza, S., and De Souza, C.R.B., 2024. "You're on a Bicycle with a Little Motor": Benefits and Challenges of using AI Code Assistants. In: *2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp.144-152.

Miah, T., and Zhu, H., 2024. User Centric Evaluation of Code Generation Tools (Invited Paper). In: *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pp.109-119.

Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., and Jiang, Z.M.J., 2023. GitHub copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, 203(C), p.111734.

Moratis, K., Diamantopoulos, T., Nastos, D.N., and Symeonidis, A., 2024. Write me this Code: An Analysis of ChatGPT Quality for Producing Source Code. In: *Proceedings - 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR 2024*. Institute of Electrical and Electronics Engineers Inc., pp.147-151.

Nazir, A., and Wang, Z., 2023. A comprehensive survey of ChatGPT: Advancements, applications, prospects, and challenges. *Meta-Radiology*, 1, p.100022.

Nikolaidis, N., Flamos, K., Gulati, K., Feitosa, D., Ampatzoglou, A., and Chatzigeorgiou, A., 2024. A Comparison of the Effectiveness of ChatGPT and Co-Pilot for Generating Quality Python Code Solutions. In: *Proceedings - 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion, SANER-C 2024*. Institute of Electrical and Electronics Engineers Inc., pp.93-101.

Niu, C., Li, C., Ng, V., Chen, D., Ge, J., and Luo, B., 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp.2136-2148.

Niu, C., Zhang, T., Li, C., Luo, B., and Ng, V., 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. In: *Proceedings - 2024 IEEE/ACM 1st International Conference on AI Foundation Models and Software Engineering, FORGE 2024*. Association for Computing Machinery, Inc., pp.103-107.

Ouyang, S., Zhang, J.M., Harman, M., and Wang, M., 2024. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology*, 34, p.42.

Paul, D.G., Zhu, H., and Bayley, I., 2024a. Benchmarks and Metrics for

Evaluations of Code Generation: A Critical Review. In: *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, pp.87-94.

Paul, D.G., Zhu, H., and Bayley, I., 2024b. ScenEval: A benchmark for scenario-based evaluation of code generation. In: *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, pp.55-63.

Petersen, K., Vakkalanka, S., and Kuzniarz, L., 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64, pp.1-18.

Petrovic, N., Konicanin, S., and Suljovic, S., 2023. ChatGPT in IoT Systems: Arduino Case Studies. In: *2023 IEEE 33rd International Conference on Microelectronics, MIEL 2023*. Institute of Electrical and Electronics Engineers Inc.

Rai, L., Khatiwada, S., Deng, C., and Liu, F., 2024. Cross-Language Code Development with Generative AI: A Source-to-Source Translation Perspective. In: *2024 IEEE 7th International Conference on Electronic Information and Communication Technology, ICEICT 2024*. Institute of Electrical and Electronics Engineers Inc., pp.562-565.

Rizvi, A., Simon, N., Tocho, J., Yongaci, A., Abi-Karam, S., and Hao, C., 2024. Evaluating Large Language Models for High-Level Synthesis. In: *2024 IEEE Opportunity Research Scholars Symposium (ORSS)*, pp.49-52.

Sakib, F.A., Khan, S.H., and Karim, A.H.M.R., 2023. *Extending the Frontier of ChatGPT: Code Generation and Debugging*. George Mason University, Virginia.

Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., and Sarro, F., 2024. A survey on machine learning techniques applied to source code. *Journal of Systems and Software*, 209, p.111934.

Siddiq, M.L., Roney, L., Zhang, J., and Santos, J.C.S., 2024. Quality Assessment of ChatGPT Generated Code and their Use by Developers. In: *Proceedings - 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR 2024*. Institute of Electrical and Electronics Engineers Inc., pp.152-156.

Su, H., Ai, J., Yu, D., and Zhang, H., 2023. An Evaluation Method for Large Language Models' Code Generation Capability. In: *Proceedings - 2023 10th International Conference on Dependable Systems and Their Applications, DSA 2023*. Institute of Electrical and Electronics Engineers Inc., pp.831-838.

Tony, C., Mutas, M., Ferreyra, N.E.D., and Scandariato, R., 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. In: *Proceedings - 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR 2023*. Institute of Electrical and Electronics Engineers Inc., pp.588-592.

Vijayaraghavan, P., Shi, L., Ambrogio, S., Mackin, C., Nitsure, A., Beymer, D., and Degan, E., 2024. VHDL-Eval: A Framework for Evaluating Large Language Models in VHDL Code Generation. In: *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, pp.1-6.

Wan, Y., Bi, Z., He, Y., Zhang, J., Zhang, H., Sui, Y., Xu, G., Jin, H., and Yu, P., 2024. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 56, p.309.

Wang, J., and Chen, Y., 2023. A Review on Code Generation with LLMs: Application and Evaluation. In: *Proceedings - 2023 1st IEEE International Conference on Medical Artificial Intelligence, MedAI 2023*. Institute of Electrical and Electronics Engineers Inc., pp.284-289.

Wang, W., Ning, H., Zhang, G., Liu, L., and Wang, Y., 2024. Rocks coding, not development: A human-centric, experimental evaluation of LLM-supported SE tasks. *Proceedings of the ACM on Software Engineering*, 1, pp.699-721.

Xiao, T., Treude, C., Hata, H., and Matsumoto, K., 2024. DevGPT: Studying Developer-ChatGPT Conversations. In: *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24. Association for Computing Machinery, New York, NY, USA, pp.227-230.

Xu, B., Nguyen, T.D., Le-Cong, T., Hoang, T., Liu, J., Kim, K., Gong, C., Niu, C., Wang, C., Le, B., and Lo, D., 2023. Are We Ready to Embrace Generative AI for Software Q and A? In: *Proceedings - 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*. Institute of Electrical and Electronics Engineers Inc., pp.1713-1717.

Yan, D., Gao, Z., and Liu, Z., 2023. A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT. In: *Proceedings - 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*. Institute of Electrical and Electronics Engineers Inc., pp.1887-1898.

Yang, Z., Liu, F., Yu, Z., Keung, J.W., Li, J., Liu, S., Hong, Y., Ma, X., Jin, Z., and Li, G., 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *Proceedings of the ACM on Software Engineering*, 1(FSE), pp.1585-1608.

Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z., and Zhang, Y., 2024. A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, 4, p.100211.

Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Liang, G., Li, Y., Wang, Q., and Xie, T., 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pretrained Models. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, pp.428-439.

Zhao, Z., Sun, J., Cai, C.H., and Wei, Z., 2024. *Code Generation Using Self-Interactive Assistant*. Institute of Electrical and Electronics Engineers (IEEE), United States, pp.2347-2352.