Revista Brasileira de Computação Aplicada, November, 2025

DOI: 10.5335/rbca.v17i3.16310
Vol. 17, Nº 3, pp. 1–13
Homepage: seer.upf.br/index.php/rbca/index

ORIGINAL PAPER

A Systematic Review about Large Language Models (LLMs) applied to Code Generation

Fabio Alecsandro Bacin[1], Braulio Adriano de Mello[1], Giancarlo Dondoni Salton[1], Samuel da Silva Feitosa[1]

[1]Computer Science - Universidade Federal da Fronteira Sul, SC-484, Km 02 - Fronteira Sul, 89815-899, Chapecó - SC - Brazil

*fabio.bacin@estudante.uff.edu.br; braulio@uffs.edu.br; gian@uffs.edu.br; samuel.feitosa@uffs.edu.br

Received: 2024-09-24. **Revised:** 2025-07-13. **Accepted:** 2025-11-01.

Abstract

Large Language Models (LLMs) for code generation represent a significant advancement in software development by boosting productivity, simplifying repetitive tasks, enabling automated testing, and promoting best practices. This paper presents a systematic literature review (SLR) of studies focused on LLMs applied to code generation. The review enhances understanding of the capabilities and limitations of these models, outlining both their benefits and challenges. The review protocol included a search on the Google Scholar database using relevant keywords related to LLMs and code generation. A total of 112 articles were initially retrieved, from which 15 were selected based on relevance and quality criteria. Of these, 8 were analyzed in depth to evaluate various approaches and outcomes, while the remaining 7 provided the theoretical foundation for the study. This work contributes to the growing body of knowledge in the field and supports future research and applications of LLMs in software engineering.

Keywords: Fuzzing; Natural Language Processing; AI in Software Engineering; Automatic Code Synthesis; Transformer Models.

Resumo

Modelos de Linguagem de Grande Escala (LLMs) aplicados à geração de código representam um avanço significativo no desenvolvimento de software, ao aumentar a produtividade, simplificar tarefas repetitivas, possibilitar testes automatizados e promover boas práticas. Este artigo apresenta uma revisão sistemática da literatura (RSL) sobre estudos focados no uso de LLMs para geração de código. A revisão aprofunda a compreensão das capacidades e limitações desses modelos, destacando seus benefícios e desafios. O protocolo de revisão incluiu uma busca na base de dados Google Scholar, utilizando palavras-chave relacionadas a LLMs e geração de código. Foram inicialmente encontrados 112 artigos, dos quais 15 foram selecionados com base em critérios de relevância e qualidade. Destes, 8 foram analisados em profundidade para avaliar diferentes abordagens e resultados, enquanto os 7 restantes forneceram a base teórica do estudo. Este trabalho contribui para o avanço do conhecimento na área e apoia futuras pesquisas e aplicações dos LLMs na engenharia de software.

Palavras-Chave: Geração de Código; Processamento de Linguagem Natural; Inteligência Artificial na Engenharia de Software; Síntese Automática de Código; Modelos Transformer.

1 Introduction

Large Language Models (LLMs) have emerged as one of the most revolutionary technologies in the field of Artificial Intelligence (AI) and Natural Language Processing (NLP). These models, trained on vast amounts of textual data, are capable of understanding, generating, and manipulating human language in highly sophisticated

ways. According to Brown et al. (2020), LLMs, such as GPT-3, demonstrate an unprecedented ability to perform a wide range of linguistic tasks, including translation, summarization, and even source code generation, without the need for task-specific training.

The underlying architecture of LLMs is typically based on transformers, an approach introduced by Vaswani et al. (2017). Transformers use attention mechanisms that allow the models to effectively handle long-term dependencies between words in a text, which is crucial for contextual understanding. This architecture has revolutionized the field of NLP, enabling models to scale to billions of parameters, as discussed by Kaplan et al. (2020). This scalability is one of the main factors that allow LLMs to achieve unprecedented levels of performance on standard NLP benchmarks.

The application of LLMs goes beyond natural language processing. Models like GPT-3 have been explored in areas such as source code generation, medical diagnosis, and even creative content creation. According to Chen et al. (2021), the ability of these models to generate code from natural language descriptions opens new possibilities for software development, potentially revolutionizing software engineering practices.

However, despite significant advances, LLMs also present challenges and limitations. One of the main problems is the tendency of these models to generate factually incorrect or biased information, as observed by Bender et al. (2021). This raises important ethical questions about the use and dissemination of AI-generated information. Additionally, the need for enormous computational resources to train these models is a significant barrier for many organizations, limiting the democratization of access to this technology.

LLMs represent a significant advancement in the field of NLP and AI, with applications extending to various areas beyond natural language. However, continuous research is essential to address the ethical and technical challenges associated with the use of these models. Future studies should focus on improving accuracy, reducing bias, and making the technology more accessible.

Source code generation through NLP is a promising area that seeks to automate significant parts of the software development process. By using natural language descriptions, these techniques can transform textual requirements into functional code, speeding up development and reducing human errors. Implementations of LLMs, exemplified by GPT-3 and Codex, can interpret and generate code in various programming languages from natural language commands. Besides LLMs, techniques such as Programming by Demonstration (PBD) and Programming by Example (PBE) are also used, where user examples and specifications guide the code generation. These models have been used in a range of tasks, including code generation, functionality explanation, interface generation, test automation, etc.

This systematic review aims to enrich the field of study by analyzing a range of scientific articles that represent the state of the art on the topic. The intention is to synthesize the findings and perspectives of various researchers to answer previously outlined research questions. By using the results and conclusions of these authors, the review seeks to identify patterns, gaps in existing knowledge, and opportunities for future investigations, thus contributing to the advancement of knowledge in the area.

The remainder of this article is organized as follows: Section 2 presents an introduction to Large Language Models (LLMs), exploring their importance and applications in the field of artificial intelligence and natural language processing. Section 3 describes in detail the Systematic Review Protocol, including the methods, inclusion and exclusion criteria, and the data sources used for article selection. Section 4 conducts a detailed analysis of the selected works, offering a critical review of the included studies, discussing the benefits, challenges, and techniques employed by LLMs in code generation. Section 5 synthesizes the main findings and conclusions drawn from the analysis of the selected articles. Finally, Section 6 presents the quantitative results of the analysis, while Section 7 provides the final considerations of this work, highlighting relevant contributions, practical implications, and suggestions for future research in the area.

2 Large Language Models (LLMs)

The creation of a Large Language Model (LLM) involves several complex stages that rely on recent advances in the field of artificial intelligence and natural language processing. These models are built using deep neural network architectures, especially transformers, which were introduced by Vaswani et al. (2017). The process can be divided into several phases, including data collection, preprocessing, model architecture, training, and evaluation.

2.1 Data Collection

The first step in creating an LLM is collecting a large amount of textual data. These data are generally obtained from various sources, including books, articles, websites, and other forms of digital text. As mentioned by Brown et al. (2020), the success of an LLM largely depends on the diversity and quality of the data used. Therefore, it is essential to ensure that the collected data are broad and representative.

2.2 Model Architecture

The model architecture is another fundamental step. Most modern LLMs use the transformer architecture, which relies on attention mechanisms to handle the long-term dependencies between words in a text. The transformer, introduced by Vaswani et al. (2017), allows the model to focus on different parts of the text with different weights, which is crucial for contextual understanding and consists of multiple layers of attention and feed-forward, where each layer processes the input in parallel, in contrast to the sequential approaches of previous models, such as LSTMs and GRUs. This approach not only increases training efficiency but also improves the model's ability to capture complex relationships within the data. The scalability

of transformers is one of the main factors that allow LLMs to achieve unprecedented performance levels on standard NLP benchmarks, enabling applications beyond NLP, including source code generation. According to Kaplan et al. (2020), the scalability of transformers allows the creation of models with billions of parameters, which significantly increases their capacity for understanding and generating text.

2.3 Model Training

Model training is probably the most resource-intensive phase in terms of computational resources. It involves feeding the model with large amounts of data and adjusting its parameters to minimize prediction error. This process can take weeks or even months and requires the use of specialized hardware, such as GPUs and TPUs. During training, techniques such as regularization and optimization are applied to improve model performance and avoid overfitting, as discussed by Radford et al. (2019).

2.4 Model Evaluation

Model evaluation is essential to ensure that it has learned correctly and can generalize well to new data. This is done using test datasets that were not used during training. Metrics such as perplexity and accuracy are commonly used to measure model performance, as explained by Linzen et al. (2021). Additionally, it is important to conduct qualitative evaluations to ensure that the model does not generate factually incorrect or biased responses.

Creating an LLM is a multi-phase process that requires a combination of high-quality data, advanced preprocessing techniques, sophisticated model architectures, significant computational resources, and rigorous evaluation methods. These components work together to produce models that can perform a wide range of linguistic tasks with high accuracy.

3 Systematic Review Protocol

Source code generation using LLMs has emerged as a promising area at the intersection of artificial intelligence and software engineering. These models, trained with vast corpora of data, have shown remarkable capabilities in generating usable code in real-world scenarios but also face significant challenges such as syntactic accuracy, type system constraints, and the detection of complex bugs in the generated code. This work aims to conduct a Systematic Literature Review on the state of the art regarding source code generation using LLMs, analyzing methodologies, results, and major contributions of selected studies in this area.

The methodology for this systematic review was divided into several stages to allow a comprehensive analysis of the selected articles.

3.1 Review Planning

This section is dedicated to describing the planning of the systematic literature review, where research questions that guided the investigation were defined, inclusion and exclusion criteria were established to delimit the scope of this study, as well as the data source, which is crucial for information collection. Additionally, keywords and synonyms that gave rise to the search string used were presented to ensure a structured and replicable approach to researching LLMs and their ability to automate and optimize source code creation.

3.1.1 *Research Questions*

The research questions establish the expected outcomes of the systematic analysis presented in this article. To study the chosen articles and understand the adopted methodologies, four specific research questions were stipulated:

- **Q1- What are the main benefits of LLMs in source code generation?**
- **Q2- What are the main challenges and limitations of LLMs in source code generation?**
- **Q3- How can LLMs be optimized to improve code generation?**
- **Q4- What are the main tools and techniques of LLMs for code generation?**

3.1.2 *Search String*

The goal of the search string in a systematic review is to enable the search in scientific databases to obtain the most relevant articles in the study area.

Keywords and Synonyms. From the research questions, it was possible to extract the main keywords related to the subject. To obtain better coverage of the articles, some synonyms were also defined for each keyword.

- Code Generation: Fuzzer, Fuzzing;
- Compiler Testing: API Testing, Automated Testing, Library Testing; and
- Large Language Model: Generative AI, LLM;

Search String. With the keywords and their synonyms defined, it was possible to elaborate the following search string, which can be used in different databases.

```
(Compile Testing OR API Testing OR Automated Testing
    OR Library Testing) AND (Code Generation
  OR Fuzzer OR Fuzzing) AND (Large Language Model
        OR Generative AI OR LLM)
```

3.1.3 *Inclusion and Exclusion Criteria*

To enable the selection of studies, a set of inclusion and exclusion criteria was established. These criteria are used to filter articles, allowing the inclusion of only research that strictly aligns with the objectives of the proposed systematic review and excluding those that do not meet predefined methodological or thematic standards.

Exclusion Criteria:

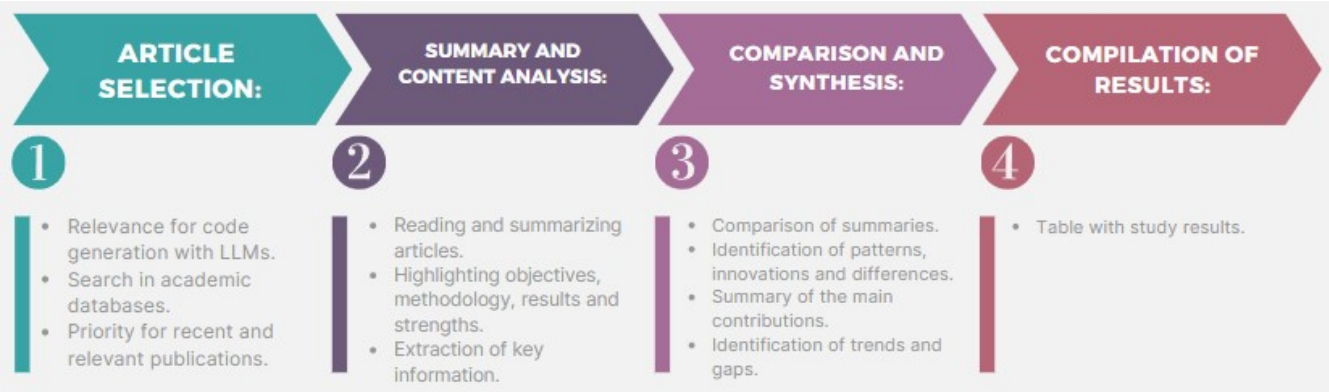

Figure 1: Systematic Review Process Flowchart.

- The study does not use LLMs in the context of code generation.
- The study does not fit the context of code generation.
- The study does not provide empirical data.
- The study is not a primary study.
- The study is duplicated.

Inclusion Criteria:

- Tools for source code generation.
- Techniques for source code generation.
- Use of LLMs for source code generation.

3.1.4 *Data Sources*

Google Scholar[1] was chosen as the data source due to its broad coverage of academic publications, including articles, theses, and technical reports. Its advanced search tools allow precise results, and free access facilitates the availability of information. The constant update of content ensures access to the latest research from different sources, making it ideal for a comprehensive and up-to-date systematic review on LLMs and code generation.

3.2 Conducting the Review

Having completed the planning, the systematic review process was initiated, as illustrated in Fig. 1. The flowchart represents the sequential and interconnected steps that were followed to conduct an analysis of the selected articles.

The first step, Article Selection, involved searching the Google Scholar database using the search string, resulting in 112 articles. In the context of the proposed study, the articles were categorized according to their relevance and suitability to the topic of source code generation using LLMs. From this initial classification, inclusion and exclusion criteria were applied to select those most appropriate for this research.

Table 1 summarizes the distribution of studies according to the mentioned criteria, divided into exclusion and inclusion:

Table 1: Criteria and Selected Studies

Criteria	Studies
Exclusion	
The study does not use LLMs in the context	36
The study does not fit the context	21
The study does not provide empirical data	23
The study is not a primary study	14
The study is duplicated	3
Inclusion	
Tools for code generation	2
Techniques for code generation	4
Use of LLM for code generation	9
Total	112

As observed, a total of 36 studies were excluded for not using LLMs in the desired context, while 21 studies did not fit the specific context or practical application analyzed. Additionally, 23 studies were discarded for not providing empirical data, being primarily theoretical or conceptual. Of the remaining articles, 14 were not primary studies, meaning they did not present original data but rather reviews, meta-analyses, or comments on other studies, and 3 other studies were considered duplicates, meaning publications that were repeated or essentially identical to others already included in the analysis.

After applying the exclusion criteria, the inclusion criteria were analyzed, where, of the selected articles, 2 studies described or analyzed specific tools developed for code generation, while 4 studies focused on methods and techniques for generating code randomly, possibly for testing or experimentation. Finally, 9 studies explored the application of LLMs in code generation.

Of the 15 selected articles, 8 were used for comparative analysis, evaluating approaches and results. The remaining 7 served as an introductory theoretical framework, providing an overview of the essential theories and concepts of the field.

In the second step, Summary and Content Analysis, described in greater detail in Section 4, each selected article was read and summarized. At this stage, the objectives, methodology, results achieved, and strengths of each study were highlighted. This detailed analysis allowed for the extraction of key information that

[1] https://scholar.google.com

contributes to a deep understanding of the approaches proposed in the articles. Section 5 synthesizes these findings and conclusions, providing a consolidated overview. Finally, in the Results Compilation step, described in Section 6, the analysis results were compiled into a table, allowing visualization of the results presented by the different studies.

4 Analysis of Selected Works

The articles selected for this systematic review address the research questions defined in the review protocol. Each study was chosen based on its relevance and contribution to understanding the impact of LLMs on code generation. The main research questions involve the benefits and challenges of LLMs in code generation, as well as how these models can be optimized to improve their effectiveness. The analysis of the selected works highlights significant advances in the area and points to future research directions.

4.1 Fuzz4All

The main highlight of Fuzz4All is its use of Large Language Models (LLMs) to automatically generate and modify test inputs, enabling fuzzing (random input generation) in various programming languages with different characteristics. It generates automatic prompts for AI, creating diverse and realistic inputs autonomously, which increases the efficiency and effectiveness of tests. This method addresses significant limitations of traditional fuzzers, which are generally language or version-specific and have limited input diversity. The study by Gao et al. (2023) presents this innovative approach to fuzzing, an automated technique used to identify flaws and vulnerabilities in software systems.

- **Universal Fuzzing:** Fuzz4All uses LLMs as engines for generating and mutating inputs, enabling universal fuzzing in multiple programming languages. Nine Systems Under Test (SUT) using six different languages (C, C++, Go, SMT2, Java, and Python) were evaluated. In all cases, it outperformed language-specific fuzzers in terms of code coverage.

- **Autoprompting for Fuzzing:** Autoprompting is a technique that creates prompts for LLMs specifically suited for fuzzing, distilling user inputs into effective prompts. This automates the creation of diverse and realistic inputs, increasing efficiency and effectiveness in random code generation.

- **LLM-powered Fuzzing Loop:** An LLM-powered fuzzing loop is introduced that iteratively updates prompts to create new fuzzing inputs, combining previously generated inputs with natural language instructions for mutations. The approach resulted in the discovery of 98 bugs in widely used systems such as GCC, Clang, Z3, CVC5, OpenJDK, and the quantum computing platform Qiskit, with 64 bugs confirmed as previously unknown.

4.1.1 *Challenges of Traditional Fuzzers*

Traditional fuzzers face three main challenges:

- Coupling with the target system and language: They are designed for specific languages, making it difficult to reuse in other languages or versions.
- Lack of support for evolution: They cannot keep up with the evolution of systems and languages, losing effectiveness in new versions.
- Limited generation capacity: Both generation-based and mutation-based fuzzers struggle to cover the entire input space of a language.

Extensive evaluation of Fuzz4All showed that it achieves superior code coverage compared to language-specific fuzzers, with an average improvement of 36.8%. Additionally, it supports directed fuzzing for specific features, proving highly effective for testing new features or components of a system. Case studies showed that Fuzz4All can generate complex inputs that previous methods could not, revealing important bugs that were confirmed and fixed by developers.

Fuzz4All represents a significant advancement in fuzzing, combining the flexibility and power of LLMs with innovative input generation and mutation techniques. Its ability to apply fuzzing universally and evolve with the tested systems makes it a valuable tool for detecting vulnerabilities in a wide range of software, ensuring greater security and reliability in software development.

4.2 VeriGen

This study, conducted by Thakur et al. (2023), involves fine-tuning pre-existing LLMs on Verilog datasets compiled from GitHub and Verilog textbooks. The goal is to assess the functional correctness of the generated code using a specially designed test set, presenting a set of custom problems and testbenches. The study used BigQuery to collect public Verilog repositories from GitHub, resulting in a training corpus of approximately 50,000 files with a total size of 300 MB after filtering. Seventy Verilog-based textbooks were downloaded from an online library, and text was extracted using OCR. After cleaning, the combined corpus of code and text totaled 400 MB. Five pre-trained models were fine-tuned, ranging from 345M to 16B parameters, including MegatronLM, CodeGen, and commercial models like GPT-3.5-turbo. Fine-tuning the models involved multiple GPUs due to high memory demand. Fine-tuning was performed for a single epoch using HPC clusters. The evaluation included two sets of problems:

- Set I: 17 Verilog problems of varying complexity, with testbenches developed to validate functional correctness.
- Set II: 181 HDLBits problems, testing a wide range of hardware design challenges and Verilog syntax.

4.2.1 *Evidence*

- **Model Performance:** The fine-tuned CodeGen-16B model outperformed GPT-3.5-turbo, demonstrating a 1.1% improvement in overall performance and a

41% improvement in syntactically correct Verilog code generation. Fine-tuned CodeGen-16B showed competitive performance, especially on intermediate and advanced complexity problems.

- **Impact of Training Data:** The study revealed that incorporating content from Verilog textbooks significantly improved the quality of the generated code. The model trained on both GitHub code and textbook content (CodeGen-2B-FT++) outperformed other models in all problem difficulties and prompt description levels.

- **Variation in Code Quality:** The quality of the generated code strongly depends on the prompt details. Detailed prompts resulted in higher quality code. The introduction of diverse data, such as educational resources, led to an improvement in model performance.

- **Emerging LLM Performance:** GPT-3.5-turbo and GPT-4 showed notable performance, especially on advanced problems, but faced challenges in specific tasks. The efficiency and capability of generating functional code of fine-tuned models like CodeGen-16B-FT were noted.

- **Potential of Fine-Tuned LLMs in Hardware Design Automation:** The results demonstrate that smaller models, fine-tuned for Verilog tasks, can compete with larger commercial models in terms of efficiency and output quality. Future improvements may involve incorporating domain-specific data and exploring hybrid approaches that combine the strengths of different LLMs.

4.3 AlphaCode

This system uses transformer-based neural networks to solve competitive programming problems, marking a significant advancement in AI's ability to generate functional code for complex problems. The study by Li et al. (2022) discusses the creation and evaluation of AlphaCode, a system developed by DeepMind for competitive code generation.

Automatic generation of programs from high-level descriptions is a challenging task in computer science. Systems capable of generating functional code have important practical applications, such as increasing programmer productivity and facilitating programming education. Traditionally, code generation has been limited to specific domains or small code snippets. However, AlphaCode represents a significant breakthrough by competing in complex programming problems.

AlphaCode was trained on a vast dataset of human code from GitHub, using an encoder-decoder transformer model. The system generates millions of code samples for each problem, filtering and clustering these samples to submit up to 10 best solutions. This process includes several enhancement techniques, such as multi-query attention, masked language modeling, tempering, conditioning, and demonstration learning.

To evaluate AlphaCode's performance, researchers used simulated competitions on the Codeforces platform. AlphaCode achieved an average ranking within the top 54.3% of human participants, an unprecedented feat for an AI system in programming competitions. This performance corresponds to a beginner programmer with a few months to one year of experience.

AlphaCode was able to solve 29.6% of the problems in the CodeContests test set with up to 10 submissions per problem. The system's scalability was a key finding, where increasing the number of generated samples led to a log-linear increase in problem-solving rates. The evaluation showed that the system could generate novel solutions for previously unseen problems, demonstrating significant understanding and reasoning in problem-solving.

4.3.1 Notable Features

- **Scalability:** The ability to generate a large number of samples and filter the best solutions was essential to AlphaCode's success.

- **Reasoning Capability:** AlphaCode demonstrated significant reasoning skills in solving complex problems, without simply memorizing training.

- **Practical Applicability:** The techniques developed for AlphaCode can be applied to improve programmer productivity and democratize access to programming.

- **Advancement in AI Research:** The system represents a significant advancement in the field of AI-generated code, opening doors for future research and applications.

AlphaCode exemplifies the potential of transformers to solve complex problems through code generation. The system's success in competitive programming competitions highlights its reasoning capabilities and the importance of scalability and effective filtering of generated samples. This work not only demonstrates the state of the art in code generation but also sets a new benchmark for future research in AI and programming.

AlphaCode is a powerful proof of concept for how AI systems can be trained to solve complex programming problems competitively. The breakthrough represented by AlphaCode has significant implications for the future of AI-assisted programming, both in terms of productivity and accessibility.

4.4 CODET

Generating code solutions for a programming problem can be enhanced with pre-trained language models like Codex, which produce multiple diverse samples. A significant challenge is selecting the most appropriate solution among the generated samples. Solutions like this are important as manually creating test cases to evaluate code quality and correctness is costly and time-consuming. The study by Nijkamp et al. (2022) presents CODET, an innovative method that uses the same pre-trained language models to automatically generate test cases, reducing human effort and increasing test coverage. CODET executes the code samples with the generated test cases and performs

a dual execution agreement, considering the consistency of results with the generated test cases and other code samples.

- **Test Case Generation:** The pre-trained model is used to generate a large number of test cases for each programming problem from an elaborated prompt. These test cases are then used to quickly verify the correctness of any generated solution.

- **Dual Execution Agreement:** Inspired by the RANSAC algorithm to select the best code solution, each of them is executed on each generated test case, forming groups of solutions that pass the same test cases. These groups are ranked based on the number of tests passed and the functional consistency of the solutions.

Experiments were conducted on four benchmarks: HumanEval, MBPP, APPS, and CodeContests, using five different pre-trained models. The results show that CODET can significantly improve performance in selecting code solutions compared to previous methods. For example, CODET increased the pass@1 metric in HumanEval to 65.8%, representing an absolute improvement of 18.8% over the code-davinci-002 model.

4.4.1 Notable Features

- **Reduction of Human Effort:** Automatic test case generation significantly reduces the need for manual creation of these cases.

- **Performance Improvement:** The dual execution agreement approach demonstrated consistent and significant improvements in selecting code solutions.

- **Versatility:** The method was tested and shown to be effective in multiple benchmarks and with different pre-trained language models.

CODET leverages the inherent power of pre-trained language models to generate both code solutions and test cases, facilitating the selection of the best solution through an efficient and automated method. Future challenges related to generating executable code and the additional computational cost for generating test cases will be explored to further improve CODET.

4.5 SynCode

Code generation by LLMs has shown remarkable capabilities but faces significant challenges, especially with syntactic errors. This problem is exacerbated in smaller models and underrepresented programming languages in training data. The presence of syntactic errors in the generated code can hinder its practical integration, causing functionality issues and debugging challenges, as discussed by Ugare et al. (2024).

SynCode is a framework that uses a programming language's grammar to create an efficient lookup table, called DFA mask store. This table is built based on the language's grammar terminals, allowing SynCode to keep only syntactically valid tokens and reject invalid ones during code generation.

- **Offline Construction:** The DFA mask store table is built offline from the regular expressions representing the language's grammar terminals.

- **Integration with LLMs:** SynCode can be combined with any existing LLM decoding algorithm, such as greedy search, beam search, or sampling.

- **Error Reduction:** In experiments with simplified context-free grammars (CFGs) for Python and Go, SynCode showed a significant reduction of 96.07% in syntax errors when combined with state-of-the-art LLMs.

- **Incremental Parsing:** SynCode uses an incremental parser that processes the partial code generated by the LLM, producing acceptance sequences and remainders. A "remainder" refers to the remaining code segment after partial parsing by the parser. When SynCode processes partial code, it accepts as much as possible of this code as syntactically correct, leaving the unaccepted part as a "remainder". These sequences are then used to generate token masks, eliminating syntactically invalid tokens. This allows SynCode to gradually correct syntax errors as the code is generated, ensuring higher accuracy and quality of the final code produced.

- **Evaluation:** Experiments were conducted with LLMs such as CodeGen, WizardCoder, and Llama, evaluated on challenging datasets like HumanEval and MBXP. The evaluation considered both LALR(1) and LR(1) as base parsers, showing that LR(1) parsers are more efficient in generating acceptance sequences.

- **Framework for Syntactic Decoding:** SynCode is a general and efficient framework for generating syntactically correct code.

- **SynCode Tool:** Implementation of SynCode that can be integrated with any language defined by a CFG.

- **Extensive Evaluation:** Performance of SynCode evaluated in code generation for Python and Go.

- **Findings:** SynCode demonstrated a significant reduction in syntax and indentation errors:
  Python: Reduction of syntax and indentation errors by over 90% compared to standard generation.
  Go: Reduction of syntax errors by over 90%, highlighting SynCode's effectiveness in underrepresented programming languages.

4.5.1 Notable Features

- Efficiency: Offline construction of the DFA mask store table, allowing fast and efficient decoding.
- Generality: Applicable to any language defined by CFG.
- Error Reduction: Substantial reduction in syntax and indentation errors, improving the accuracy of generated code.

SynCode represents a significant advancement in LLM code generation, providing an efficient way to ensure syntactic correctness. This framework can be particularly useful for underrepresented programming languages in model training data, improving the quality of generated code and facilitating its practical integration.

4.6 Tricky Bugs

Detecting bugs in software systems is a growing challenge in software engineering, especially with the prevalence of automatically generated code by generative AI. The study presented by Liu et al. (2024) addresses the problem of identifying tricky bugs in programs that pass existing tests but may still contain difficult-to-detect defects.

4.6.1 Relevant Contributions

i. Automated Input Diversity (AID):

- Combination of LLMs and Differential Testing: AID combines LLMs with differential testing to generate test inputs that reveal failures and oracles for plausibly correct programs.
- Generation of Program Variants: AID generates variants of the program under test to capture different behaviors.
- Filtering and Validation: Existing test cases are used to filter program variants and ensure the generation of accurate test oracles.

ii. Extensive Evaluation:

- Datasets: Evaluation conducted on two large datasets with tricky bugs: TrickyBugs and EvalPlus.
- Better Performance: AID outperforms state-of-the-art methods, such as Differential Prompting Plus (DPP), in recall, precision, and F1-score.

4.6.2 Process and Result

The AID approach follows three main steps:

i. Generation of Program Variants: LLMs are used to generate variants of the program based on the program's specifications. This involves creating different versions of the program that theoretically should meet the same specifications, allowing the testing of various possible scenarios.
ii. Generation of Test Inputs: Test inputs are generated from specific generators to ensure the diversity and legality of the inputs. This ensures that the inputs used in the tests are varied enough to cover different use cases while remaining within the acceptable limits defined by the program's specifications.
iii. Differential Testing: Differential tests are conducted to identify inconsistencies in the outputs of the variants. A majority voting principle is used to determine the accuracy of the results, where the majority of similar results are considered correct, helping to detect defects in the program variants.

Preliminary studies indicated that the accuracy of test cases generated directly by ChatGPT was low, around 6.3%. Most errors (92.2%) were due to incorrect test oracles, highlighting the need to combine LLMs with more robust testing methods to improve accuracy.

Compared to conventional methods, the AID approach demonstrated significant improvements, especially in programs with complex bugs. AID achieved an F1-score of 85.09% on some datasets, substantially outperforming baseline tests.

Using diversity in differential tests, rather than following the majority voting principle, proved effective in detecting defects. This implies that variety in test inputs can reveal more flaws than simple vote counting.

- **Combination of Techniques:** Integrating LLMs with differential testing improves the accuracy of test oracle generation, making the process more efficient.
- **Comprehensive Evaluation:** Using large datasets and comparison with multiple baselines validate AID's effectiveness, demonstrating its robustness in various scenarios.
- **Comprehensive Evaluation:** Using large datasets and comparison with multiple baselines validate AID's effectiveness, demonstrating its robustness in various scenarios.
- **Superior Results:** AID demonstrates significant improvements in all evaluated performance metrics, standing out as a more effective approach to detecting complex bugs.

The article presents an innovative approach that combines the natural language understanding power of LLMs with the differential testing technique to enhance bug detection in software. AID proved superior to existing methods, offering a more robust and accurate solution for identifying defects in plausibly correct programs.

4.7 CODE4STRUCT

The research presented by Wang et al. (2022) addresses the CODE4STRUCT model, an innovative proposal for event argument extraction (EAE) using code generation. This model, developed by researchers at the University of Illinois, Urbana-Champaign, explores the capability of LLMs trained with a combination of text and code to translate natural language into code structures.

The central goal of the study is to investigate how translating semantic structures into code can improve structured prediction tasks in NLP. Specifically, the study focuses on event argument extraction, formulating EAE as a code generation problem, which allows the use of programming language features such as inheritance and type annotations to introduce external knowledge or add constraints.

The CODE4STRUCT approach involves converting event type ontologies into Python class definitions. Using sufficient contextual examples, EAE is treated as a code generation problem, where the model is trained to instantiate events based on provided sentences. This methodology allows the model to use programming language features to impose argument constraints and

leverage event hierarchies.

- **Comparison with Supervised Models:** In the referenced article, the CODE4STRUCT study demonstrated that even using only 20 training examples per event type, its performance is comparable to supervised models trained with 4,202 instances. This method outperformed the current state of the art in few-shot learning datasets, achieving an absolute gain of 29.5% in the F1-score metric. This approach highlighted the efficiency and robustness of the model, even under limited training conditions.

- **Generalization and Data Efficiency:** The model demonstrated significant data efficiency, achieving performance comparable to fully supervised methods with far fewer annotated examples. Additionally, the event hierarchy allows resource-less event types to leverage training examples from related event types, improving prediction for event types with no training data.

- **Advantages of Code Generation:** Formulating EAE as a code generation problem proved advantageous over text-based prompt variants. Using features such as type annotations and default argument values naturally imposes argument constraints for output structures.

- **Performance in Different LLMs:** Experiments with different LLMs showed that the CODE4STRUCT model is robust and maintains superior performance in various configurations, especially when sufficient contextual examples are provided.

4.7.1 Positive Aspects

i. Innovation in Methodology:

- Using code generation for event argument extraction is an innovative approach that leverages advances in LLMs trained on code and text corpora.

ii. Data Efficiency:

- The model achieves superior results using a minimal amount of training data, standing out for its efficiency.

iii. Flexibility and Applicability:

- The ability to use event hierarchies and examples from related event types significantly expands the model's applicability, especially in low-resource scenarios.

The study demonstrates that the CODE4STRUCT approach is effective for structured prediction tasks in NLP, offering a promising alternative to traditional text-based methods. With robust results and data efficiency, this methodology has the potential to be applied in a variety of complex structured prediction tasks in the future. The success of CODE4STRUCT in outperforming supervised models with less data highlights its relevance and potential impact in the field of NLP.

4.8 Expectation vs. Experience

The study presented by Vaithilingam et al. (2022) explores the usability of GitHub Copilot, a code generation tool that uses LLMs. Conducted by Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman, the work investigates how programmers interact with Copilot compared to IntelliSense, the standard code completion tool in Visual Studio Code.

The research involved a study with 24 participants, including undergraduate, master's, Ph.D. students, and a software engineer. Participants completed programming tasks in Python with and without the aid of Copilot. The tasks varied in difficulty (easy, medium, and difficult). Data collection included screen recordings, audio recordings, and responses to pre and post-task questionnaires.

The results showed that:

- Copilot users completed fewer tasks than IntelliSense users.
- Although Copilot did not significantly reduce task completion time, it was preferred by most participants (19 out of 24).
- Participants found Copilot more useful than IntelliSense (6.16 vs. 4.45 on a scale of 1 to 7).

Copilot was seen as a good starting point for programming tasks, especially useful for tasks that users did not know how to start. However, participants struggled to understand and modify the code generated by Copilot, leading to a perception of loss of control and concerns about code reliability.

Participants used Copilot as a substitute for internet searches, although this led to overconfidence in the generated code, resulting in less validation and more time spent debugging incorrect code.

When encountering errors in the generated code, participants generally tried to fix the code but often found it difficult due to a lack of understanding. Some preferred to rewrite the code completely rather than try to repair it.

Three main obstacles were identified:

i. Difficulty in understanding and evaluating the correctness of the generated code.
ii. Underestimation of the effort needed to fix bugs.
iii. Ambiguity and sensitivity when using comments as specifications for Copilot.

4.8.1 Critical Analysis

Participants preferred Copilot for daily tasks, despite not significantly reducing task completion time. Suggestions for improvements include:

- Providing multiple code suggestions for comparison.
- Integrating online searches to validate the generated code.
- Providing explanations and comments in the generated code to facilitate understanding and debugging.

The study reveals that while Copilot offers a useful starting

point for programming tasks, understanding and fixing the generated code remain significant challenges. Future improvements should focus on support for validation, task decomposition, and integration with online resources to increase confidence and effectiveness of programmers using LLM-based code generation tools.

5 Synthesis of Selected Works

This section presents a synthesis of the main findings, methodologies, and conclusions drawn from the analysis of each scientific article selected for this systematic review.

Gao et al. (2023) introduceed Fuzz4All, a method that uses LLMs to generate and mutate inputs in fuzzing, applicable to multiple programming languages. Evaluated in nine systems under test, Fuzz4All outperformed language-specific fuzzers in terms of code coverage. The autoprompting technique and the iterative fuzzing loop allowed the discovery of numerous bugs, highlighting the method's effectiveness.

Thakur et al. (2023) conducted a study with LLMs fine-tuned on Verilog data, focusing on the functional correctness of the generated code. The study used public repositories from GitHub and textbooks. The evaluation included problems of varying complexity and demonstrated that models like CodeGen-16B, when fine-tuned, can outperform commercial models like GPT-3.5-turbo, especially in generating syntactically correct Verilog code.

Li et al. (2022) discussed the development of AlphaCode, a system that uses transformers to solve competitive programming problems. Trained with a vast dataset of human code from GitHub, AlphaCode demonstrated the ability to compete in programming competitions, achieving results comparable to human programmers. The system stood out for its scalability and reasoning skills, showing a significant advancement in code generation.

Nijkamp et al. (2022) presented CODET, which uses pre-trained LLMs to automatically generate test cases and select code solutions. Inspired by the RANSAC algorithm, the method showed significant improvements in selecting code solutions, reducing human effort and increasing test coverage. Evaluations in various benchmarks demonstrated CODET's effectiveness in improving performance in solution selection.

Ugare et al. (2024) introduced SynCode, a framework that uses programming language grammar to ensure syntactic correctness during code generation. Evaluated in Python and Go, SynCode showed a significant reduction in syntax and indentation errors. The offline construction of the DFA mask store table allows for fast and efficient decoding, standing out as a general method applicable to any language defined by context-free grammar.

Liu et al. (2024) addresses detecting tricky bugs in plausibly correct programs through AID. Combining LLMs and differential testing, AID generated program variants and test inputs, demonstrating superiority over traditional methods. The evaluation showed significant improvements in recall, precision, and F1-score, highlighting the approach's effectiveness in detecting complex bugs.

The study by Wang et al. (2022) presents the CODE4STRUCT model, which uses LLMs trained with text and code for event argument extraction (EAE). The methodology involves converting event type ontologies into Python class definitions, treating EAE as a code generation problem. Key findings indicate that CODE4STRUCT, even with few training examples, outperforms traditional supervised models and demonstrates data efficiency, using programming features to impose constraints and improve event prediction.

Vaithilingam et al. (2022) explores the usability of GitHub Copilot compared to IntelliSense. The study involved 24 participants performing programming tasks in Python with and without Copilot. The results indicated that Copilot was preferred by most participants, despite not significantly reducing task completion time. The main challenges include difficulty understanding and evaluating the correctness of the generated code, underestimating the effort needed to fix bugs.

The reviewed studies demonstrate the growing versatility and impact of large language models (LLMs) in code generation, testing, and debugging across diverse programming contexts. From enhancing fuzzing processes and detecting complex bugs to generating syntactically correct domain-specific code and competitive programming solutions, the findings highlight LLMs' capacity to improve coverage, accuracy, and efficiency in software development tasks. Approaches such as grammar-guided generation, fine-tuning on specialized datasets, and iterative test-driven loops usually outperform traditional or language-specific methods, while also reducing human intervention. However, usability studies reveal persistent challenges in code comprehension and error assessment, indicating that while LLMs are powerful accelerators, their effective integration requires complementary human oversight and robust evaluation mechanisms.

6 Results: Answers to Research Questions

Based on the detailed analysis of the selected articles, the results were described according to the previously defined research questions.

- Q1- What are the main benefits of LLMs in source code generation?
  - LLMs increase efficiency and speed in software development, simplify repetitive processes, and promote the adoption of best practices. Models like GPT-3 can understand and generate code from natural language descriptions, automating significant parts of the programming process. This reduces human errors and facilitates the maintenance of consistent and high-quality code.
- Q2 - What are the main challenges and limitations of LLMs in source code generation?
  - LLMs face challenges such as generating incorrect or biased information and the need for enormous

computational resources for training. Additionally, problems with syntactic accuracy, type system constraints, and detecting complex bugs are common. Studies highlight ethical and practical issues that need to be addressed to improve the use of these models.

- Q3 - How can LLMs be optimized to improve source code generation?
  - Incorporating specialized content, such as textbooks and high-quality code repositories, can improve the accuracy and functionality of the generated code. Techniques such as automatic test case generation, seen in the CODET method, help validate and select the best code solutions. Models like SynCode, which use grammar to ensure syntactic correctness, are also promising. Continuous research should focus on improving data efficiency, generation accuracy, and the ability to adapt to different programming domains.
- Q4 - What are the main tools and techniques of LLMs for source code generation?
  - The LLMs for source code generation mainly use the transformer architecture. Among the most notable tools are GitHub Copilot, which uses the Codex model to suggest and complete code snippets as developers type, and AlphaCode, which can solve complex programming problems and compete with human programmers. Innovative techniques include CODET, which automatically generates test cases and selects the best code solutions, inspired by the RANSAC algorithm, and SynCode, which ensures syntactic correctness during code generation using programming language grammar. Another significant tool is VeriGen, fine-tuned specifically for Verilog code generation, demonstrating high accuracy and functionality, outperforming commercial models. These tools and techniques, based on the transformer architecture, represent significant advances in the efficiency and functionality of software development, highlighting the transformative potential of LLMs in the programming field.

6.1 Quantitative Results

The Table 2 is a detailed table providing a comprehensive analysis of eight studies on the application of Large Language Models (LLMs) in code generation. Each row represents a study, detailing the study name, authors, objective, methodology, dataset used, and main results.

The Fuzz4All study by Gao et al. (2023) uses LLMs for input generation and mutation in fuzzing, applying the method to multiple programming languages. Data were collected from GitHub and Verilog textbooks, resulting in a 36.8% improved code coverage and the discovery of 98 bugs.

The VeriGen study by Thakur et al. (2023) fine-tuned LLMs with Verilog data from GitHub and textbooks, improving overall performance by 1.1% and generating correct code by 41%.

The AlphaCode study by Li et al. (2022) uses transformers to solve competitive programming problems, training with GitHub data. This study achieved a top 54.3 ranking in Codeforces competitions, solving 29.6% of the problems.

The CODET study by Nijkamp et al. (2022) automatically generates test cases and uses the RANSAC algorithm to select code solutions, improving the pass@1 metric to 65.8%, an absolute increase of 18.8%.

The SynCode study by Ugare et al. (2024) ensures syntactic correctness during code generation using Python and Go grammars. This resulted in a 96.07% reduction in syntax errors.

The Tricky Bugs study by Liu et al. (2024) combines LLMs and differential testing to detect bugs, evaluating on TrickyBugs and EvalPlus datasets, achieving an F1-score of 85.09%.

The CODE4STRUCT study by Wang et al. (2022) focuses on argument extraction from events using code generation, converting event type ontologies into Python class definitions, surpassing the state of the art by 29.5% in F1-score with limited data.

Finally, Expectation vs. Experience by Vaithilingam et al. (2022) investigates the usability of GitHub Copilot, comparing it with Intellisense, revealing that although Copilot was preferred by most participants, it did not significantly reduce task completion time. This study involved participants of varying experience levels performing programming tasks in Python.

This detailed analysis highlights significant advances in the use of LLMs for code generation, demonstrating improvements in accuracy, efficiency, and error detection capability, despite challenges such as fixing syntax errors and the need for intensive computational resources.

7 Conclusion

This systematic review examined the application of Large Language Models (LLMs) in code generation, highlighting their potential, benefits, challenges, and optimization methods. The analysis of eight selected studies demonstrated significant advances in the area, contributing to understanding how LLMs can transform software development practices. By evaluating different approaches and results, the review provides a comprehensive overview of the current state of the art and identifies opportunities for future research and innovation.

The benefits of LLMs in code generation are evident in their ability to increase efficiency, automate repetitive tasks, and promote the adoption of best practices. However, challenges such as syntactic accuracy, type system constraints, and bug detection require continuous research and development to improve the reliability and functionality of the generated code.

Optimizing LLMs involves incorporating high-quality, diverse data and developing innovative techniques such as automatic test case generation and syntactic error correction. Tools like Fuzz4All, VeriGen, AlphaCode, CODET, SynCode, Tricky Bugs, CODE4STRUCT, and

Table 2: Analysis of the eight studies on the application of Large Language Models (LLMs) in code generation.

Study	Authors	Objective	Methodology	Database and benchmarks	Main Results
Fuzz4All	Gao et al. (2023)	Use of LLMs for input generation and mutation in fuzzing	Applying fuzzing in multiple programming languages	GitHub, Verilog textbooks	Improved code coverage by 36.8%; discovered 98 bugs
VeriGen	Thakur et al. (2023)	Fine-tuning LLMs on Verilog data	Collecting Verilog data and fine-tuning LLM models	GitHub, Verilog textbooks	CodeGen-16B improved 1.1% in overall performance and 41% in correct code generation
AlphaCode	Li et al. (2022)	Using transformers to solve competitive programming problems	Training with GitHub human code dataset	GitHub	Top 54.3% in Codeforces competitions; solved 29.6% of problems
CODET	Nijkamp et al. (2022)	Automatic generation of test cases and selection of code solutions	Generating test cases and using RANSAC algorithm	Various benchmarks: HumanEval, MBPP, APPS, CodeContests	Improved pass@1 to 65.8%, absolute increase of 18.8%
SynCode	Ugare et al. (2024)	Ensuring syntactic correctness during code generation	Building DFA mask store table and using grammar	GitHub, Python and Go datasets	Reduction of syntax errors by 96.07%
Tricky Bugs	Liu et al. (2024)	Detection of tricky bugs in plausibly correct programs	Combination of LLMs and differential testing	TrickyBugs EvalPlus	Improved recall, precision, and F1-score; F1-score of 85.09%
CODE4STRUCT	Wang et al. (2022)	Event argument extraction using code generation	Using Python class definitions for events	Event ontologies, contextual examples	29.5% higher F1-score compared to the state of the art with limited data
Expectation vs. Experience	Vaithilingam et al. (2022)	Exploring the usability of GitHub Copilot	Study with 24 participants performing tasks with and without Copilot	Participants of varying experience levels	Preferred by 19 participants; perceived as more useful than IntelliSense

GitHub Copilot demonstrate the transformative potential of LLMs in software development, offering practical solutions and new approaches to code generation.

Future research should focus on addressing the ethical and technical challenges associated with LLMs, improving their accessibility, and expanding their applications in various programming domains. By advancing the understanding and capabilities of LLMs, the field of software engineering can continue to evolve, benefiting from the innovative potential of these powerful models.

References

Bender, E. M., Gebru, T., McMillan-Major, A. and Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big?, *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, ACM, pp. 610–623. https://doi.org/10.1145/3442188.3445922.

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, *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Curran Associates Inc., Red Hook, NY, USA. https://dl.acm.org/doi/abs/10.5555/3495724.3495883.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. et al. (2021). Evaluating large language models trained on code, *arXiv preprint arXiv:2107.03374* . https://doi.org/10.48550/arXiv.2107.03374.

Gao, Y., Yang, X., Yu, Y., Zhou, Z., Chen, H. and Zhang, T. (2023). Fuzz4all: Universal fuzzing with large language models, *arXiv preprint arXiv:2308.04748* . https://arxiv.org/abs/2308.04748.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J. and Amodei, D. (2020). Scaling laws for neural language models, *arXiv preprint arXiv:2001.08361* . https://doi.org/10.48550/arXiv.2001.08361.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., Freitas, N. d., Kavukcuoglu, K. and Vinyals, O. (2022). Competition-level code generation with alphacode, *Science* **378**(6617): 1092–1097. https://www.science.org/doi/10.1126/science.abq1158.

Linzen, T., Dupoux, E. and Goldberg, Y. (2021). The unreasonable effectiveness of recurrent neural networks in natural language processing, *Proceedings of the National Academy of Sciences* **118**(24): e2107151118.

Liu, K., Liu, Y., Chen, Z., Zhang, J. M., Han, Y., Ma, Y., Li, G. and Huang, G. (2024). Llm-powered test case generation for detecting tricky bugs, *Proceedings of the Conference*, ACM, ACM, Washington, DC, USA. Available at https://arxiv.org/html/2404.10304v1.

Nijkamp, E. et al. (2022). Codet: Code generation with generated tests, *arXiv preprint arXiv:2207.10397v2* . https://arxiv.org/abs/2207.10397v2.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. and Sutskever, I. (2019). Language models are unsupervised multitask learners, *OpenAI Blog* **1**(8). Available at https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Thakur, S., Ahmad, B., Pearce, H., Tan, B., Dolan-Gavitt, B., Karri, R. and Garg, S. (2023). Verigen: A large language model for verilog code generation, *ACM Transactions on Design Automation of Electronic Systems* . https://doi.org/10.48550/arXiv.2308.00708.

Ugare, S., Suresh, T., Kang, H., Misailovic, S. and Singh, G. (2024). Improving llm code generation with grammar augmentation, *arXiv preprint arXiv:2403.01632v1* . https://arxiv.org/abs/2403.01632v1.

Vaithilingam, P., Zhang, T. and Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models, *CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI '22 Extended Abstracts)*, ACM, p. 7. https://doi.org/10.1145/3491101.3519665.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I. (2017). Attention is all you need, *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, Curran Associates Inc., Red Hook, NY, USA, p. 6000–6010. https://dl.acm.org/doi/10.5555/3295222.3295349.

Wang, X., Li, S. and Ji, H. (2022). Code4struct: Code generation for few-shot event structure prediction, *arXiv preprint arXiv:2210.12810* . https://doi.org/10.48550/arXiv.2210.12810.