Analysis of LLM Code Synthesis in Software Productivity

Anurag Anand, Shivali Chopra, Mohit Arora
School of Computer Science and Engineering Lovely Professional University Phagwara Punjab, India
Corresponding author: Anurag Anand, Email: anuraganand7618@gmail.com

The use of LLMs in code generation tools has introduced a paradigm shift in software development, streamlining the process and enhancing automation and efficiency. This study presents a comprehensive analysis of the applications and effectiveness of the Large Language Model (LLM) in code synthesis based upon the analysis of various models. The LLM techniques where programming codes are significantly constrained on high level and low-level programming paradigm, has emerged as a dominant strategy in software productivity due to its inherent ability to promote efficiency and minimize time to build logic. Our research systematically explores the impact of LLM on the performance outcomes on various programming languages, comparing it to traditional code practices. We analyze multiple case studies, quantitatively evaluating the success rates, efficiency, and problem-solving capacity of LLM-based solutions. Preliminary findings indicate that LLM encourages a unique problem-solving approach, despite its limitations, often results in highly efficient and innovative solutions. However, the technique also presents a steep learning curve that may deter novice programmers. This study aims to contribute to the body of knowledge on software productivity strategies and the continuing discourse on code efficiency and optimization.

Keywords: Code Generation, DevinAI, GPT-4, Code Llama, StarCoder, Tabnine, DS Code Generation, LLM Security, LLEMMA.

2025. In Mukesh Saraswat & Rajani Kumari (eds.), *Applied Intelligence and Computing*, 247–259. Computing & Intelligent Systems, SCRS, India. DOI: https://doi.org/10.56155/978-81-955020-9-7-24

1 Introduction

Language models (LMs) are computational models that assign probabilities to sequences of tokens [1], commonly utilized in natural language processing tasks. Recently, there has been a growing interest in applying LMs to model source code written in programming languages [2], [3], [4], [5], [6]. These code language models have demonstrated exceptional performance in various tasks such as code completion and generating code from natural language descriptions. The latest advancements in large language models tailored for various tasks. Notably, one of the largest models in this domain, Codex [7] has been integrated into the real-world development tool through GitHub Copilot [8], functioning as an in-IDE developer assistant that autonomously generates code based on the developer's context.

In recent years, these models have proven their effectiveness across various software engineering tasks, including test generation, documentation generation, and even synthesizing functional programs from natural language descriptions. Emerging products like GitHub Copilot, Amazon CodeWhisperer, and Tabnine, which leverage code generation models [9], are gaining popularity among developers. Despite many models being trained on multiple programming languages, evaluations often focus solely on Python. Python's widespread adoption among machine learning researchers has led to the creation of several benchmarks and datasets, making it the primary language for evaluation. However, expanding evaluations to include other languages is crucial to support a broader range of programmers [7]. In this article, we aim to address this gap by presenting analysis of benchmark performed by renowned organizations on various programming languages.

Code completion tools [10] and code generation[2], [3], [4], [10], [11] tools are pivotal for optimizing developer efficiency and improving the quality of software development outputs. Code completion tools, such as IntelliSense in Visual Studio Code (https://code.visualstudio.com/) or the built-in code completion in JetBrains IDEs (https://www.jetbrains.com/), are designed to provide contextually relevant suggestions of variables, fields, methods, types, and other code snippets. These tools, available in most code editors, allow developers to minimize grammatical and logical errors, reduce keystrokes, and explore new Application Programming Interfaces (APIs) without the need for mental context switching to external documentation tools or API browsers.On the other hand, code generators fundamentally differ from code completion tools. With the advent of advanced natural language processing technologies. LLMs have emerged, offering a wide range of applications, particularly in code generation. Code generators actively leverage LLMs, taking the programmer's input, processing it through the specified models, and returning the output to the programmer's workspace. However, unlike code completion tools, code generator's outputs are not locally produced; But in first quarter of 2024 OpenDevin [12] has been released which also has integrated terminal, environment, as well as online browser for reading the latest libraries available from its source. It [13] can generate more extensive outputs, including lines or blocks of code capable of building function bodies or other programming constructs.

The growing interest in AI-assisted code generators has led to the propagation of unverified information about their capabilities and performance. This includes assessing the code generators on multiple dimensions: Code Validity, Code Correctness, Code Security, Code Reliability, and Code Maintainability [14] [15]. So our aims is to provide a comprehensive understanding of their usefulness and potential pitfalls.

In this study, we undertake a systematic analysis of prevailing code models – Codex [7] , ChatGPT [16], GPT-Neo[17], Code Llama [11] , and PaLM [18] etc across diverse programming languages, aiming to elucidate the code modeling design decisions. These models are subject to rigorous comparative analysis and evaluation against existing models employing the HumanEval [19] benchmark and an unseen evaluation dataset across many programming languages.

This study addresses various use case in paradigm shifting of software productivity:

1 Impact of LLM on Code Completion Efficiency.
2 Analysis of Performance in LLM-Based Code Refactoring Tools

3 Impact of LLM Integration on Software Development Automation
4 Cybersecurity Risks and Mitigation in LLM-Synthesized Code
5 Implications of LLM-Based Code Generation for Novice Programmers
6 Contribution of LLM Tools to Software Development Productivity
7 Future Directions and Challenges in LLM-Based Code Tools

2 Literature Survey

This section presents an outline of the diverse activities integral to software productivity that can be facilitated by LLMs. Constructed from the context provided, these tasks encompass a broad range of programming-related operations that leverage the capabilities of LLMs to optimize the process of software development.

2.1 Code Completion

Lozhkov A et. al. presented BigCode [3] scientific project for responsible development of Code LLMs and build StarCoder2 [20] with Software Heritage (https://www.softwareheritage.org/), consist of 619 programming language accompanied with high quality of data from Code Documentation, GitHub and Kaggle Notebooks. One prominent contender, DeepSeekCoder-33B [21], has demonstrated remarkable performance in high-resource programming languages. However, for low-resource languages, StarCoder2-15B [3] has emerged as a formidable competitor, either matching or surpassing DeepSeekCoder-33B [21] in code completion benchmarks. However, concerns regarding the adequacy of hidden tests and ambiguity in problem descriptions have led to the introduction of the EvalPlus [3] framework. This framework addresses these issues by significantly increasing the number of tests in benchmarks like HumanEval+ [19] and MBPP+ [3]. Additionally, MultiPL-E [17], a multi-language benchmark, provides a comprehensive evaluation platform by translating HumanEval problems into various programming languages. Furthermore, the evaluation extends beyond traditional code completion tasks to encompass data science tasks in Python, as demonstrated by the DS-1000 [22] benchmark. StarCoder2- 3B, -7B, and -15B [3] models exhibit varying degrees of proficiency in completing these tasks, with StarCoder2-15B [3] emerging as the top performer, even surpassing larger models such as CodeLlama-13B and CodeLlama-34B [11].

Studies have emphasized the importance of accuracy metrics such as pass@k, which measure the model's ability to produce correct code solutions within the top K number of predictions.(see Table 1)

Table 1. PASS@1result on MULTI-PLE [17], evaluated at temp – 0.2 and top-p – 0.95

Model	C++	C#	Go	Java	JS	Perl	Swift	TS
StableCode-3B [23]	28.4	14.4	19.3	27.8	32	9.4	13.2	29.6
DeepSeekCoder-1.3B [18]	28.3	21.3	19.1	29.2	28.3	12.5	11	27.4
StarCoderBase-3B [3]	19.4	13.3	13.3	19.2	21.3	11.3	10	22.8
StarCoder2-3B[3]	27.2	20.5	23.6	27.4	35.4	13.6	25.1	34.4
CodeLlama-7B[11]	26.4	21	20.9	28.2	31.6	16.9	24.9	33.4
DeepSeekCoder-6.7B[18]	46.7	32.9	31	39.7	46.6	30.4	30.3	39.5
StarCoderBase-7B [4]	23.3	19.3	19.6	24.4	27.4	15.2	15.1	27.5

StarCoder2-7B[3]	33.6	20.7	20.2	29.4	35.4	16.6	26.1	36.3
CodeLlama-13B [11]	37.4	24.8	26.6	37.5	39.3	23.4	30.1	40.1
StarCoderBase-15B[4]	30.6	20.6	21.5	28.5	31.7	16.3	16.7	32.1
StarCoder2-15B[4]	41.4	29.2	26.2	33.9	44.2	37.2	34.2	43.8
CodeLlama-34B[11]	41.4	30.7	28.7	40.2	41.7	28.5	35.3	42.1
DeepSeekCoder-33B [18]	51.2	35.3	34.2	43.8	51.3	31	35.8	48.4

2.2 Code Refactoring

Code refactoring involves restructuring class designs without altering external behavior. Techniques includes extracting classes, renaming, encapsulating fields, polymorphism, interface extraction, composition, breaking dependency cycles, and removing duplicate code. The goal is to enhance maintainability, extensibility, and readability while preserving functionality [9].

Studies conducted for tools like Amazon CodeWhisperer [24], ChatGPT [6] and GitHub Copilot [8] evaluated their performance based on various metrics such as correctness, validity, reliability, security, and maintainability. The study aimed to assess the effectiveness of these tools in generating code solutions and their adaptability to different input parameters. They [9] utilized the HumanEval [19] problem dataset to evaluate the performance of each tool. (see Table 2)

Table 2. Comparative Analysis of tools based on their features and performance metrics.

Tool	Correct (%)	Partially Correct (%)	Incorrect (%)	Average Time for Refactoring (minutes)
GitHub Copilot [9]	20-22	26-27	50-53	9.1
Amazon CodeWhisperer [9]	18-20	25-26	54-56	5.6
ChatGPT [9]	61.6	25.6	12.8	8.9

Correct (%): Indicates the percentage of problems for which the tool generated correct code solutions.
Partially Correct (%): Represents the percentage of problems for which the tool produced partially correct code solutions.
Incorrect (%): Shows the percentage of problems for which the tool generated incorrect code solutions.
Average Time for Refactoring (minutes): Reflects the average time taken to eliminate code smells or improve maintainability, measured in minutes.

Introducing dummy function names affected the success rate of code generation tools [2], [9], [10], [11]. ChatGPT exhibited the highest percentage of correct solutions among the three tools. Providing accurate and clear problem descriptions is crucial for improving the performance of code synthesis. Code refactoring techniques such as extracting classes, renaming, encapsulating fields, and removing duplicate code play a significant role in enhancing code maintainability, extensibility, and readability. Amazon CodeWhisperer and GitHub Copilot demonstrate rapid improvements [9], indicating their potential for various coding tasks in the future.

2.3 Comment Generation / Documentation

Santa Coder [2], [3], [4], [10], [11] investigates the impact of various data filtering strategies on the performance of BigCode [25], which is LLM trained for code generation tasks. One such filtering strategy involves analyzing the comments-to-code ratio in Python, Java, and JavaScript files. The approach utilizes modules like AST (Abstract Syntax Tree) and tokenize for Python files and Pygments (https://pygments.org/) for Java and JavaScript files to extract comments and docstrings The analysis reveals codebase contain no comments, approximately 20% for Python and Java files; and 45% for JavaScript files. A minimum threshold of 1% is applied, removing an additional 3% of files in each language [25]. Files with a comment-to-code ratio above 80% are considered to have poor quality and are filtered out, resulting in the elimination of 2% of data across all language [25]. Performance metrics (pass@1, pass@10, pass@100) for different language models under various data filtering is computed. For instance, in Java, the Comments-to-code ratio filtering method improves Pass@1 from 0.1 to 0.11 compared to the same baseline models [25]. Similar improvements are observed in JavaScript and Python models. They compares the performance of BigCode Santa [4] variants with 1.1B parameters against the other model, noting that variants trained on more Python data tend to perform better. Specifically, the stars variant, trained with 32% Python data, outperforms the tokenizer fertility variant, which only includes 28.5% Python data.

2.4 Security in Code LLM

LLMs has significantly impacted various cognitive tasks, this progress has raised substantial concerns regarding cybersecurity risks associated with the code synthesized by these models. In response, a sophisticated cybersecurity safety measurement suite named CYBERSECEVAL [14] has been developed to address two primary cybersecurity risks posed by LLMs and provide actionable insights for risk mitigation.

The first major risk involves the potential for LLM-generated code to deviate from established security best practices, thereby introducing vulnerabilities. Empirical evidence, such as GitHub's CoPilot and Meta's CodeCompose [26] studies, indicates that a notable portion of code suggested by LLMs may contain exploitable weaknesses. CYBERSECEVAL addresses this risk by seamlessly integrating into the development and testing workflows of LLM designs [9] [14]. It leverages a robust Insecure Code Detector (ICD) framework [14] comprising a knowledge base of static analysis rules. These rules, designed to detect insecure coding practices as defined by the Common Weakness Enumeration (CWE) standard, facilitate the identification and assessment of vulnerabilities in LLM-generated code across various programming language, which includes the identification of 189 patterns related to 50 CWEs across eight commonly used programming languages: C#, C++, Rust , JS, Python, Java, C and PHP. By iteratively refining LLM models based on the insights garnered from CYBERSECEVAL [14], developers can enhance the security posture of the code produced by their AI systems.

The second prominent risk pertains to the possibility of LLMs inadvertently facilitating malicious activities related to computer systems [9], [14], [26]. Although foundational LLMs are generally programmed to resist aiding in illicit activities, CYBERSECEVAL [14] investigates whether this resistance extends to coding-enabled models when confronted with openly malicious requests. Through meticulous evaluation of LLM responses to test cases crafted to simulate cyberattacks, CYBERSECEVAL [14] offers invaluable insights into potential misuse scenarios. This enables LLM designers to proactively identify and mitigate risks associated with the malevolent application of their models.

CYBERSECEVAL's [14] comprehensive approach to cybersecurity safety measurement encompasses two primary methodologies:

1 Insecure Code Detection: This component employs a sophisticated knowledge base of 189 static analysis rules aligned with the CWE’s standard. Leveraging these rules,

CYBERSECEVAL [14] automatically generates test prompts by identifying instances of insecure coding practices in real-world open-source codebases. Subsequently, it evaluates LLM responses to determine adherence to or deviation from established security best practices.

2 Compliance Evaluation with Cyberattacks: CYBERSECEVAL [14] crafts test cases by manually authoring prompt fragments designed to solicit LLM assistance in executing cyberattacks as per industry-standard frameworks like MITRE ATTACK. These test cases are then used to evaluate LLM responses, assessing their compliance with requests involving various cyberattack tactics, techniques, and procedures.

Assessing the cybersecurity safety of LLM completions through automated means poses considerable challenges. static analysis methodology [14] attains a manually validated precision of 96% and recall of 79% in identifying instances of insecure code generation by LLMs. Moreover, approach used for detecting malicious LLM completions for cyberattack assistance achieves a precision of 94% and a recall of 84% [14].

2.5 Code Generation

Code generation, essentially the process of automatically generating code from various inputs, has evolved significantly in recent years. It has moved from simple compiler construction to more complex processes involving intelligence, mathematics, software-engineering, and advanced algorithms. This section provides a detailed review of the most recent advancements in the field of code generation, focusing on cutting-edge models such as GPT-4 [16], Codex [7], Lamma [11], StarCoder [3] etc. These models represent the future of automated programming, leveraging the power to revolutionize the software development process.

2.5.1 Code Generation

Study performed [25] evaluates the Python code-writing capabilities of eCodex [7], a GPT language model fine-tuned on GitHub code. It compares eCodex's performance with GPT-3 and GPT-J using the HumanEval evaluation set [6], [25], focusing on functional correctness in synthesizing programs from docstrings. eCodex [7] outperforms both GPT-3 and GPT-J, solving 28.8% of the problems. The study also explores the efficacy of repeated sampling, finding that with 100 samples per problem, eCodex [7] solves 70.2% of problems. Limitations of eCodex include challenges with docstrings describing long chains of operations and binding operations to variables. This research investigates the performance of various language models in code generation tasks. They [3], [4], [10] analyze models such as PaLM [18], PaLM-Coder [18], PaLM 2-S [18], StarCoder Base [3], StarCoder Python [3], and StarCoder [3], focusing on their accuracy metrics at different prediction thresholds. (see Figure 1)

Collected performance data for various language models, including PaLM, PaLM-Coder, PaLM 2-S, StarCoder Base [3], StarCoder Python [3], and StarCoder prompted [3] and many more from their respective research studies. Performance metrics such as pass@1, pass@10, and pass@100 were evaluated to assess the models' accuracy in code generation tasks.(see Figure 2)

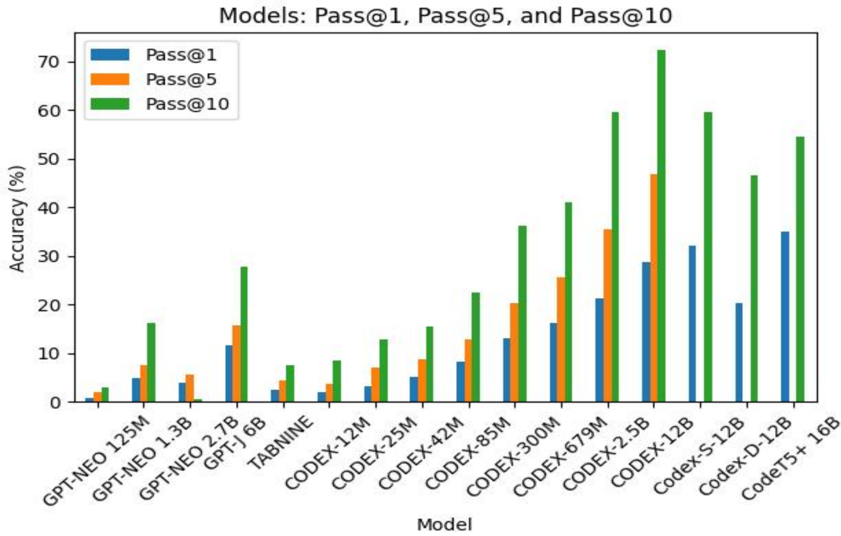

Figure 1. Pass@k evaluation on GPT-NEO, TABNINE, CODEX, T5+ [3] [2].

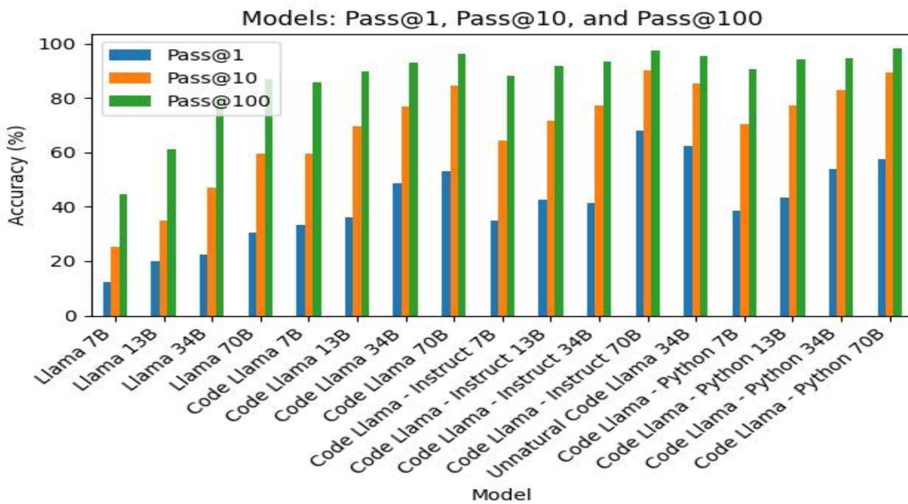

Figure 2. Pass@k evaluation on Llama with their varient models [11]

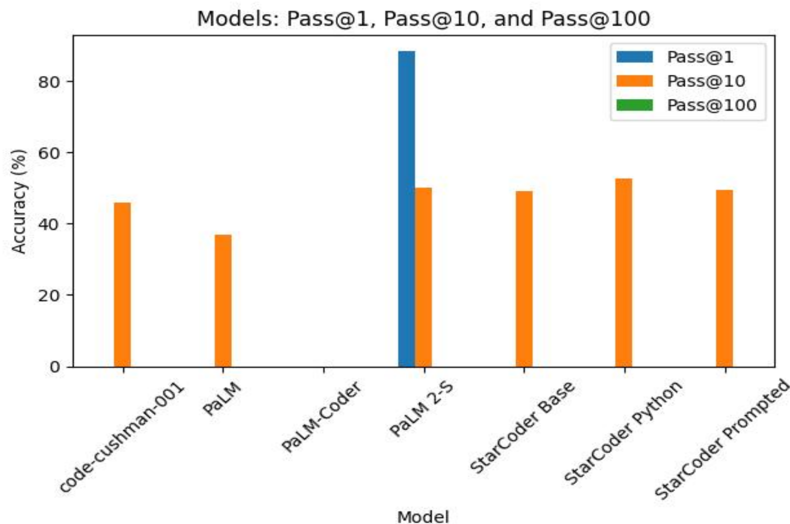

Figure 3. Pass@k evaluation on CodeCushman, PaLM, StarCoder [3], [8], [10]

This analysis revealed notable variations in the performance of the evaluated language models. PaLM [18] and its variants exhibited high accuracy at lower prediction thresholds, with pass@1 rates exceeding 80%. However, the StarCoder [3] models demonstrated competitive performance at higher prediction thresholds, achieving pass@100 rates above 40%. While the Code Llama Series model achieves above 90% at threshold of pass@100. These results underscore the trade-offs between precision and recall in code generation tasks. (see Figure 3)The study concludes by discussing broader impacts of deploying advanced code generation technologies, including considerations of safety, security, and economic implications. This research underscores the potential of eCodex [7] in generating functional Python code from docstrings while highlighting areas for improvement and the broader implications of such technology deployment.

2.5.2 Mathematical Code Generation

Mathematical problem-solving has long been a central focus of AI researcher. With the introduction of LLMs, significant step forward in this domain designed for mathematical tasks. [10], [27] are trained using a mixture of scientific studies, web data containing mathematics, and mathematical code. LLEMMA [27] is specifically trained for mathematical task forming the Proof-Pile-2 dataset while Gemini [10] is multimodal capable of performing versatile task. Through rigorous evaluation on multiple mathematical benchmarks, including MATH, GSM8k, OCWCourses, MMLU-STEM, and SAT [11] across a spectrum of mathematical reasoning tasks, LLEMMA [27] observe varying levels of performance among different language models. The initial LLEMMA 2 - 7B [11] model shows modest results, with percentages ranging from 3.2% on MATH to 29.9% on MMLU-STEM [4], [10]. The proprietary Minerva 8B [28] model demonstrates higher proficiency, particularly notable on MMLU-STEM [4], [10] with 35.6%. However, its performance is surpassed by the release of LLEMMA 7B [11], which shows significant enhancements across all tasks, notably achieving 53.1% on SAT and 37.7% on MMLU-STEM. As the model capacity increases to Code Llama 34B, further improvements are evident, with notable gains on GSM8k, MMLU-STEM, and SAT tasks. Yet, it is with the introduction of LLEMMA 34B [11] that the most substantial advancements are observed, with performance peaks of 71.9% on SAT and 51.5% on GSM8k. Comparatively, the Minerva [29] models with 62B and 540B

parameters demonstrate competitive results, especially on tasks like MMLU-STEM [27] and MATH, with scores reaching 53.9% and 33.6%, respectively. These findings underscore the incremental improvements achieved through model scaling and refinement, culminating in LLEMMA's [11] remarkable proficiency in mathematical problem-solving tasks. In contrast, Gemini Ultra [10] exhibits strong performance on elementary exams and competition-grade problem sets, it achieves 94.4% accuracy on the GSM8K benchmark [27] and outperforms competitors on middle- and high-school math competitions as well as American Mathematical Competitions , solving 32% of the questions.

2.5.3 Data Science / ML Code Generation

DS-1000 [22] groundbreaking benchmark comprising data science problems encompassing seven essential Python libraries: NumPy, Pandas, TensorFlow, PyTorch, SciPy, Scikit-learn, and Matplotlib are meticulously curated from real-world scenarios sourced from StackOverflow, ensuring they reflect diverse, practical, and realistic use cases. This diversity enriches the benchmark, providing a comprehensive evaluation framework for code AI generation models. In evaluation, they assessed the performance of various popular code generation models, including Codex, CodeGen, and InCoder,[7] using DS-1000 [22]. The results revealed a performance range from 7.4% to 43.3%, with Codex-002 [7] emerging as the most accurate model. This leaves ample room for improvement and underscores the significance of DS-1000 as a benchmark for evaluating code generation on Data Science. This assessment evaluates its effectiveness on –

- Problem Origin: Shows where problems come from.
- Surface Perturbation: Measures changes to problem presentation to prevent memorization.
- Semantic Perturbation: Assesses alterations to problem meaning for added complexity.
- Difficult Rewrite: Indicates modifications to make problems more challenging.
- Percentage of Surface-Form Constraints: Reflects constraints on solution appearance.
- Average Test Cases: Quantifies test cases used for solution verification.
- Average Problem Words: Shows average problem length.
- Average Lines of Code in Context: Reveals context length for problem-solving.
- Average Lines of Code in Solution: Displays average solution length.

2.5.4 API Integrated and Terminal Integrated Code Generation

DevinAI [12] emerges as a pioneering initiative, driven by the ambition to replicate an autonomous AI software engineer renowned for its capacity to execute engineering tasks and actively engage in software development collaborations. This project embarks on a journey to replicate, innovate, and expand upon Devin's capabilities through the collective source of intelligence. At its core DevinAI [13] has embedded with tools including shell, code editor, and web browser. Due to access to internet, it automates though the stable version of library to empower the Software building process. These tools when interacting with LMMs like GPT or Code Generation based LLMs, empower Devin to navigate the complexities inherent in software development processes with remarkable efficiency. OpenDevin [https://github.com/OpenDevin/OpenDevin] is another project which thrives on community engagement, fostering an environment where developers, researchers, and enthusiasts converge to explore the vast potential of LLMs in practical software engineering scenarios seeks to identify, dissect, and address both the strengths and limitations of Devin's original model, thus paving the way for significant advancements in the realm of AI-driven software development.

3 LLM Programming Dataset

Dataset serves as a cornerstone for various AI tasks like code generation, refactoring, comment generation, code vulnerability analysis etc [30]. A comprehensive collection of annotated programming samples is instrumental in facilitating rigorous model training and validation. This allows algorithms to

effectively generate, refactor, and augment codebases, thereby fostering a surge in innovation and automation within the realm of software development processes. Most of dataset are curated from GitHub [https://github.com] is a prime repository, hosting a vast pool of open-source projects, contributing to the wealth of programming languages and coding styles, Stack Overflow[https://stackoverflow.com/] offers the advantage of pairing code snippets with natural language descriptions, which is particularly beneficial for tasks related to code comment generation and understanding code semantics., CodeSearchNet Corpus [https://github.com/github/CodeSearchNet/] from GitHub, provides a rich dataset featuring millions of code snippets from diverse open-source projects, Conala Corpus [https://conala-corpus.github.io/] provides pairs of natural language questions with corresponding code snippets., Pile Dataset [https://pile.eleuther.ai/] employed for training models like GPT-Neo, Google's BigQuery Public Datasets [https://cloud.google.com/ bigquery/public-data /github] and py150 [https://www.sri.inf.ethz.ch/ py150] for tasks related to Python code generation and refactoring.

4 Analysis of Research Undertaken

Efficiency of LLM-Based Code Completion compared to Traditional Methods

Contextual Understanding: Traditional methods may lack the ability to understand the context in which the code is written, which can limit their code completion capabilities. The LLM approach, on the other hand, leverages the inherent capability of language models to understand long term dependencies, thus potentially offering more relevant and context-aware code completions [2], [29], [31].

Adaptability: Conventional code completion methods might struggle to adapt to new programming paradigms or styles. However, an LLM approach can continuously learn from new data, making it potentially more adaptable to evolving coding practices and trends[10], [12], [31].

Performance Metrics: To make a precise comparison, we would need to define clear performance metrics. These could include the accuracy of code completions, the time taken to suggest completions, the relevance of suggested completions, and the model's ability to handle complex code structures. Such metrics are BLEU [32], ROUGE [33], METEOR [7], [34] and CODEBLEU[25] etc.

Analyzing principal determinants of performance in LLM-Based code refactoring tools and their impact on code correctness, reliability, and maintainability

To examine this, we have considered tools like GitHub Copilot, Amazon CodeWhishperer and ChatGPT several key factors:

Training Data Quality and Diversity: The performance of LLM-based tools is heavily reliant on the quality and diversity of the code data they are trained on. The volumes of programming languages, coding styles, and problem domains covered in the training data can significantly influence the tools ability to generate correct and reliable code solutions[7], [9] .

Model Architecture and Complexity: The architecture of the underlying language model, like the number of layers, the size of the hidden states, and the type of attention mechanisms used, can impact the performance of the tool. More complex models may potentially grasp intricate code patterns better, but they may also be more prone to overfitting and require more computational resources [17], [25].

Hyperparameter Tuning: The process of hyperparameter tuning can significantly impact the performance of LLM-based tools. Optimal settings for parameters like learning rate, batch size,

dropout rate, and others can greatly affect the learning efficiency and the quality of the generated code. But tuning is too costly [21].

All these tools like GitHub Copilot, Amazon CodeWhisperer, and ChatGPT requires a systematic evaluation approach; tool's ability to generate syntactically and semantically correct code. Automated testing tools can be used to evaluate the correctness of the generated code. Consistency in generating correct solutions across a variety of tasks and programming languages can be assessed by examining the readability, modularity, and extensibility of the generated code. Tools that generate clean, well-structured, and well-documented code would score high on maintainability.

Investigating LLMs in Automating Mathematical and Data Science/ML Code Generation
In the context of mathematical code generation, [22] these tools can be instrumental in automating algorithm implementation, reducing the manual effort involved in translating mathematical concepts into executable code. For example, LLMs can generate code for complex mathematical operations, matrix computations, statistical models, or numerical methods given high-level descriptions or mathematical notations. This can speed up the development process, minimize the risk of manual coding errors, and enable developers to focus on higher-level design and problem-solving aspects. They can also facilitate the use of best practices in machine learning code development, such as ensuring reproducibility, proper data handling, and efficient resource utilization. By generating boilerplate code and automating routine tasks, these tools can enhance productivity, improve code quality, and make machine learning development more accessible to non-experts.

Cybersecurity Risks Associated with LLM's Synthesized Code
The integration of LLMs into code generation and refactoring tools introduces novel cybersecurity risks. While these models can generate syntactically correct code, they may inadvertently introduce security vulnerabilities due to a lack of understanding of secure coding practices. For instance, they might generate code with buffer overflows, insecure random number generation, or improper error handling, which can be exploited by malicious actors. LLMs could be misused to generate malicious code, such as malware or ransomware. The ability of these models to generate code based on high-level prompts could be exploited to automate the creation of harmful software, posing a significant cybersecurity risk. CYBERSECEVAL is designed to evaluate the security of synthesized code by detecting insecure code practices [9], [14]. It can identify potential vulnerabilities and provide recommendations for secure coding practices from which scenarios can be anticipated and mitigated by implementing robust monitoring and control mechanisms.

Challenges and Learning Curve of Novice Programmers Using LLM
Code generation tools can have profound implications for novice programmers. Expose beginners to various coding styles and paradigms, facilitating a broader understanding of programming. Understanding and debugging the generated code could pose significant challenges if the code is not adequately commented. while LLMs can generate syntactically correct code, they might not always adhere to best coding practices. Novice programmers risk picking up bad habits if they uncritically adopt the generated code [1], [3], [4], [14].

Trade-offs in terms of Efficiency, Innovation, and Learning Curve
LLM-based tools can aid in the learning and adoption of new programming languages or libraries, as they can provide relevant code suggestions based on a vast repository of learned patterns. This can potentially lead to faster development cycles, especially when dealing with unfamiliar coding environments [10], [11], [12]. However, the use of LLM-based tools also comes with trade-offs. While they can improve efficiency by automating code generation, they may at times produce complex or non-intuitive code that could be difficult to understand or debug, especially for less experienced developers. This could potentially increase the time spent on debugging and code comprehension, affecting overall productivity.

Future Directions and Challenges in the Development and Adoption of LLM-Code Synthesis

To scale the adaptation of LLMs to more specialized domains such as embedded systems, high-performance computing, or quantum computing. Expanding LLMs to cater to multi-modal inputs (e.g., combining natural language prompts with visual cues) is another potential growth area [10]. Adapting LLMs to more specialized domains necessitate the collection and curation of high-quality, domain-specific datasets. One of the main challenges is ensuring the security and reliability of the generated code [22]. Another challenge is the potential over-reliance on these tools, leading to a superficial understanding of programming principles among developers. To address this, it's important to develop usage guidelines for these tools, emphasizing their role as an aid rather than a replacement for a deep understanding of programming.

5 Conclusion

This study provides in-depth understanding of the complex domain of code generation. We delve into the latest advancements and sophisticated models in the field, such as Gemini, Llamma, code Cushman, PaLM, StarCoder, GPT-Neo, Codex, TabnineT5+. Also perform the analysis on various task related to programming paradigm, explored the untouched analysis of generation of ML and DL codes through LLM; analyzed various online datasets. Raises the concern in code generation vulnerability. These models leverage large-scale pre-training on code repositories to learn the statistical patterns and relationships within code. The utilization of these advanced models has shown promising results in generating high-quality code with minimal human intervention. As technology continues to evolve, the future of code generation holds immense potential for streamlining software development processes.

References

[1] C. Akiki et al., “BigScience: A Case Study in the Social Construction of a Multilingual Large Language Model.” [Online]. Available: https://hf.co/bigscience/bloom

[2] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, “CodeT5+: Open Code Large Language Models for Code Understanding and Generation,” May 2023, [Online]. Available: http://arxiv.org/abs/2305.07922

[3] A. Lozhkov et al., “StarCoder 2 and The Stack v2: The Next Generation,” Feb. 2024, [Online]. Available: http://arxiv.org/abs/2402.19173

[4] L. Ben Allal et al., “SantaCoder: don’t reach for the stars!,” Jan. 2023, [Online]. Available: http://arxiv.org/abs/2301.03988

[5] Z. Feng et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” Feb. 2020, [Online]. Available: http://arxiv.org/abs/2002.08155

[6] J. Y. Khan and G. Uddin, “Automatic Code Documentation Generation Using GPT-3; Automatic Code Documentation Generation Using GPT-3,” 2022, doi: 10.1145/3551349.

[7] S. Lu et al., “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation,” Feb. 2021, [Online]. Available: http://arxiv.org/abs/2102.04664

[8] A. M. Dakhel et al., “GitHub Copilot AI pair programmer: Asset or Liability?,” Jun. 2022, [Online]. Available: http://arxiv.org/abs/2206.15331

[9] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, “Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT,” Apr. 2023, [Online]. Available: http://arxiv.org/abs/2304.10778

[10] Gemini Team et al., “Gemini: A Family of Highly Capable Multimodal Models,” Dec. 2023, [Online]. Available: http://arxiv.org/abs/2312.11805

[11] B. Rozière et al., “Code Llama: Open Foundation Models for Code,” Aug. 2023, [Online]. Available: http://arxiv.org/abs/2308.12950

[12] W. Scott, “Devin, the first AI software engineer,” 2024.

[13] “https://www.cognition-labs.com/introducing-devin.”

[14] M. Bhatt et al., “Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models,” Dec. 2023, [Online]. Available: http://arxiv.org/abs/2312.04724
[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks,” Sep. 2019, [Online]. Available: http://arxiv.org/abs/1909.03496
[16] OpenAI, “GPT-4 Technical Report,” Mar. 2023, [Online]. Available: http://arxiv.org/abs/2303.08774
[17] F. Cassano et al., “MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation,” IEEE Transactions on Software Engineering, vol. 49, no. 7, pp. 3675–3691, Jul. 2023, doi: 10.1109/TSE.2023.3267446.
[18] A. Chowdhery et al., “PaLM: Scaling Language Modeling with Pathways,” Apr. 2022, [Online]. Available: http://arxiv.org/abs/2204.02311
[19] M. Chen et al., “Evaluating Large Language Models Trained on Code,” Jul. 2021, [Online]. Available: http://arxiv.org/abs/2107.03374
[20] R. Li et al., “StarCoder: may the source be with you!,” May 2023, [Online]. Available: http://arxiv.org/abs/2305.06161
[21] D. Guo et al., “DeepSeek-Coder: When the Large Language Model Meets Programming -- The Rise of Code Intelligence,” Jan. 2024, [Online]. Available: http://arxiv.org/abs/2401.14196
[22] Y. Lai et al., “DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation.” [Online]. Available: https://ds1000-code-gen.
[23] N. Pinnaparaju et al., “Stable Code Technical Report,” Apr. 2024, [Online]. Available: http://arxiv.org/abs/2404.01226
[24] “AWS announces Amazon CodeWhisperer (Preview).”
[25] M. Chen et al., “Evaluating Large Language Models Trained on Code,” Jul. 2021, [Online]. Available: http://arxiv.org/abs/2107.03374
[26] V. Murali et al., “AI-assisted Code Authoring at Scale: Fine-tuning, deploying, and mixed methods evaluation,” May 2023, [Online]. Available: http://arxiv.org/abs/2305.12050
[27] Z. Azerbayev et al., “LLEMMA: AN OPEN LANGUAGE MODEL FOR MATHEMATICS.” [Online]. Available: https://github.com/EleutherAI/math-lm
[28] A. Lewkowycz et al., “Solving Quantitative Reasoning Problems with Language Models,” Jun. 2022, [Online]. Available: http://arxiv.org/abs/2206.14858
[29] A. Lewkowycz et al., “Solving Quantitative Reasoning Problems with Language Models,” Jun. 2022, [Online]. Available: http://arxiv.org/abs/2206.14858
[30] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, “An Empirical Study of Code Smells in Transformer-based Code Generation Techniques,” Institute of Electrical and Electronics Engineers (IEEE), Jan. 2023, pp. 71–82. doi: 10.1109/scam55253.2022.00014.
[31] T. Oh, S. Chung, B. Lunt, R. McMahon, and R. Rutherfoord, “The roles of IT education in IoT and data analytics,” in SIGITE 2017 - Proceedings of the 18th Annual Conference on Information Technology Education, Association for Computing Machinery, Inc, Sep. 2017, pp. 39–40. doi: 10.1145/XXXXXXX.XXXXXXX.
[32] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: a Method for Automatic Evaluation of Machine Translation.”
[33] C.-Y. Lin, “ROUGE: A Package for Automatic Evaluation of Summaries.”
[34] A. Lavie and A. Agarwal, “Meteor: An Automatic Metric for MT Evaluation with High Levels of Correlation with Human Judgments,” 2007.