ANALYZING AUTOMATIC CODE GENERATION FOR LEARNING MODELS IN GENERATIVE AI

Alok Jain
Cybersecurity Engineer,
10mind.ai (Advisor)
California, USA
alok@10mind.ai

*P. William[1, 2]
[1]Deaprtment of Information Technology, Sanjivani College of Engineering, Kopargaon
[2]Amity University Dubai, UAE
william160891@gmail.com

Firas Tayseer Ayasrah
College of Education, Humanities and Science
Al Ain University
Al Ain, UAE
firas.ayasrah@aau.ac.ae

G. Prasanna Lakshmi
School of Computer Science and Engineering, Sandip University,
Nashik, Maharashtra
prasannalakshmigandi@gmail.com

Tarun Dhar Diwan
Assistant Professor and Controller of Examination (COE)
Atal Bihari Vajpayee University,
Bilaspur, India
tarunctech@gmail.com

2024 11th International Conference on Software Defined Systems (SDS) | 979-8-3315-1832-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/SDS64317.2024.10883930

***Abstract*— Independent code generation models produced by generative AI provide a new way to software development. These models automatically generate code using machine learning based on input samples. This study examines the fundamentals, applications, problems, and future prospects of AI-related automated code generation technologies. Model-based software, domain-specific code, and testing procedures are examples of these research topics. Performance analysis and assessment are used to assess the efficacy, efficiency, and reliability of several automated code generating methods. The fact that these models have pros and cons and room for development is highlighted.**

Keywords—Automatic code generation, Generative AI, machine learning, software development, testing methodologies, model-based development, domain-specific code generation.

I. INTRODUCTION

Automatic code generation models are the most cutting-edge technologies for expediting software development and stimulating generative artificial intelligence (AI) innovation. These models may hasten complicated software system creation and minimise human developer workload. Machine learning allows models to produce code from input requirements, requests, or samples [1]. Automated code creation in its simplest form might transform software development. It frees engineers from tedious coding and replaces them with AI-powered tools. Thus, engineers may concentrate on more complex design principles and problem-solving methods. Given this, its benefits are obvious. These models may boost productivity, enhance code quality, and save product launch time. Testing, debugging, and prototyping are automated in the software development lifecycle to achieve this. Due to advances in deep learning, natural language processing, and program synthesis, autonomous code generating models are gaining popularity. These models may analyse plain English descriptions, user requests, or code segments and generate code that meets certain constraints. Models that generate code may be used to develop code in many programming languages and areas, from basic scripts to complicated algorithms. Automated code production has pros and cons in real-world software development [2]. A lot of information must be considered to produce accurate and useful code. Correctness, maintainability, and codebase compatibility are among these factors. To address the ethical consequences of AI-driven automation, such as algorithmic discrimination and job displacement, thorough investigation and mitigating methods are needed. The result may be job loss.

In our study of autonomous code generation models in generative artificial intelligence, we examine its fundamentals, applications, problems, and potential future directions. Automated code creation is changing software engineering. This discovery might boost creativity and AI-driven software implementation. A paradigm shift has created fascinating new choices. This category includes code synthesis, program induction, intelligent coding assistants, and auto-completion tools. The goal of manual code testing is to ensure that the code works as intended. During testing, it's common to not compare code to design [3]. However, during automated code creation, the model is compared to its criteria, and dynamic testing may ensure that the code developed matches the executable model (Figure 1). Since both the model and code are executable, they may be excited separately using the same inputs. Subsequently, the outputs of the model and the generated code are compared with respect to certain acceptance criteria. This comparison often reveals technical challenges, such as quantization errors that result in non-identical outputs between the model and the generated code. To address this, sophisticated signal comparison methods must be applied.

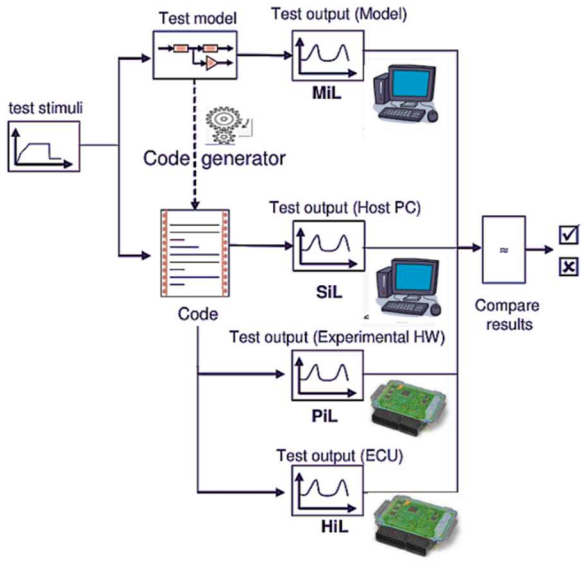

FIGURE 1: PROCESS FOR TESTING AUTOMATICALLY GENERATED CODE

The question of what constitutes appropriate test stimuli for model and code testing is fundamental. The use of structural testing criteria on both the model level (model coverage) and the code level (code coverage) for test stimuli determination is now widespread in practice. Model coverage complements the benefits of code coverage by controlling the test depth and detecting coverage holes in given test suites. Test vector generators may also generate model and code coverage test stimuli automatically. For code coverage, the Evolutionary Test Tool can produce test stimuli, whereas Reactis can generate model coverage stimuli [4,12]. The generators are classified as test vector generators. Reproducing the model and code developed at various stages of development is one of the biggest benefits of model-based development (see Figure 1). Model and code protection is achieved using many simulation methods.

- **Model-in-the-Loop (MiL):** In Model-in-the-Loop testing, the code and model must be compared every time. Here, the written code is run only in the simulation environment that houses the model. This approach lets developers thoroughly test code in a controlled environment. This ensures that the code meets model criteria and works as planned.
- **Software-in-the-Loop (SiL):** Software-in-the-loop testing approaches are used to test written code on a target software platform, such as an operating system or framework. The hardware components are not needed for this testing technique to work. Developers may test code functionality and performance in a virtual software environment before deploying it on hardware. This approach makes this assessment possible. In other words, this helps people understand code behaviour.
- **Processor-in-the-Loop (PiL):** The processor-in-the-loop testing approach includes SiL testing. Because it uses the device's CPU in the testing procedure. This allows developers to evaluate code performance and verify it meets requirements in a more realistic situation. Executing the code on the processor used in the final goods may achieve this goal.
- **Hardware-in-the-Loop (HiL):** The "hardware-in-the-loop" testing method inspects code written on the hardware it will be installed on. This technique offers the most realistic testing environment available, allowing engineers to examine code behaviour in real-world settings. This suggests that the strategy provides the most accurate testing environment. HiL testing may help you find and fix hardware compatibility and integration problems, as intended.

II. Review Of Literature

This study examines whether artificial intelligence (AI) technology outperforms human-written computer code in accuracy, efficiency, and maintainability. Computer program code developed by humans and artificial intelligence was evaluated using criteria including time and space complexity, runtime, and memory utilisation. We also assessed maintainability using the total amount of lines of code, cyclomatic complexity, Halstead complexity, and maintainability index [5,11]. We generated Java, Python, and C++ code using generative artificial intelligence for our experiments. These exams were done to fix problems on leetcode.com, a competitive coding website. Each generative artificial intelligence has to write 18 programming codes for six LeetCode tasks. GitHub Copilot, which was driven by Codex (GPT-3.0), proved to be the most successful option; CodeWhisperer was unable to fix any of the issues that were encountered. It was successful in resolving nine out of the eighteen issues, which is fifty percent of the total. BingAI Chat (GPT-4.0) was responsible for the creation of seven issues, which accounts for 38.9% of the total. ChatGPT (GPT-3.5) and Code Llama (Llama 2) were responsible for the creation of four problems, which accounts for 22.2% of the total. StarCoder and InstructCodeT5+ were responsible for exactly one problem, which accounts for 5.6% of the total. It is remarkable because ChatGPT was the only generative artificial intelligence that was able to successfully complete a difficult coding challenge, despite the fact that it only produced four accurate program codes. In conclusion, about twenty-six percent (20.6%) of the responses to the relevant question are generated by artificial intelligence. There are eleven AI-generated erroneous codes, which accounts for 8.7% of the total, and it is possible that the problem may be fixed with just minor modifications to the computer code. Therefore, the amount of time required to construct the computer code from scratch would be reduced by 8.9% to 71.3% as a consequence of this [6].

Within the scope of our study, we examine the ways in which large language models may generate natural language. In the next step, we put the information that we have obtained to use by developing two unique kinds of educational materials that are often used in instructional programming classes. We use OpenAI Codex as the primary language model to develop programming exercises and code explanations, and we assess these exercises and explanations using both objective and subjective criteria. These explanations and activities provide test cases as well

as examples of how to answer different questions. According to the results of our inquiry, a significant portion of the work that is generated digitally seems to be inventive, reasoned, and, in some instances, suitable for immediate use. In the process of feeding the model with keywords, we have seen that it is relatively easy to have an impact on the programming concepts and contextual themes that are included into the jobs. The results of our study indicate that massively generative machine learning models have the potential to become an indispensable training tool for educators [7]. However, supervision is still necessary in order to ensure that the information that is created is of a high quality before it is sent to the pupils. Following that, the implications of OpenAI Codex and other related tools for teaching foundational programming are discussed, along with planned research initiatives that have the potential to improve the quality of learning for both teachers and students.

Programming languages have been the focus of a number of research in the field of computer science ever since the field was first established. A unique paradigm in programming is now being developed, and it is being used in addition to the traditional programming languages, which include procedural, object-oriented, and functional languages. That is the very essence of what natural language programming is all about. We are of the opinion that the use of computers in natural language will release human expressiveness, in contrast to computer languages, which impose stringent syntactical limits. The purpose of this article is to investigate a number of different approaches that can automatically generate source code based on a description written in natural language. In addition to this, we categorize the techniques based on the representations of the items that are input and output. In conclusion, we take a look at the current trend of techniques and provide some suggestions for the future course of this research topic [8,13]. Our goal is to improve the process of automated code creation by using natural language. This ought to be done in order to improve the overall quality of the outcome. We came to the conclusion that the most important area of research that needs to be done is the modification of language models inside the domain of source code. This conclusion was based on the findings of the study. In addition, we believe that more efficient source code representations need to be investigated, which should include pre-trained models and embedding techniques. It has been shown that these models have demonstrated excellent performance on issues concerning the understanding of natural language.

Throughout the history of computer education, the introductory programming course has been the subject of a significant amount of research. At this point in time, there are already a number of AI-driven code development solutions that are both practical and open-source [9]. This brings possibilities as well as difficulties that are immediately available. In this position paper, we argue that the community need to act quickly in order to ascertain what opportunities may and ought to be taken advantage of, in addition to making an effort to overcome or eliminate any hurdles that may be present. Efforts should also be taken to eliminate or drastically reduce any impediments that may be present. It is presumed that these technologies will continue to grow more and more effective as they continue to increase in their widespread use. If education professionals do not respond swiftly, wisely, and collectively, they will end up missing out on opportunities to have an impact on the future as well as hurdles that they will have to face. With the assistance of this study, we would want to initiate this discussion on the education of computer users [10].

III. Principle of Automatic Code Generation in AI

Autonomous code production is a new AI breakthrough. One of the latest breakthroughs. Using machine learning, this technology eliminates the need for human programming. This technique allows automated code generation. This hypothesis is based on the idea that artificial intelligence (AI) can understand complex problems and write code to solve them. If developers employ artificial intelligence to automatically write code, they may be able to handle more difficult jobs in many areas, speed up software production, and reduce human error. These advantages may be obtained if developers build programs using AI. Machine learning models may learn from data and write code to do particular tasks. This capacity is the foundation of AI's autonomous code development. This method needs training models on massive datasets to find data patterns and connections. This ensures intended effects. In this course, the purpose is to produce code that can solve problems related to the topic being covered. Model goals and AI system capabilities will decide code type. This code might be simple scripts or complex algorithms.

To build autonomous programs, artificial intelligence must use abstraction. Because the subject remains relevant. Abstraction allows AI models to understand abstract concepts and transfer them into code that can be executed without explicit programming instructions. This is feasible because abstraction permits it. Artificial intelligence's ability to abstract complex processes into code offers automated code creation an advantage. Competitive advantage comes from this expertise. Engineers may focus on problem-solving while artificial intelligence handles implementation minutiae. This paper examines AI-related autonomous code creation foundations. Research will concentrate on autonomous code development. Additionally, the study examines the merits and cons of employing machine learning methods to generate code in a range of industries. Additionally, the paper examines these approaches' prospective applications. We discuss the ethical issues that arise when intelligent robots develop code independently. Another problem we address. Training data bias and code creation accountability are issues in this domain. Other issues include. Autonomous code, enabled by artificial intelligence, has changed software evolution on a macro level. This change is widespread. This transition allows for new productivity and creative methods.

Executable models are crucial to model-based development of control and function systems. This technique uses models to show the system's development from idea to design and implementation. This evolution is depicted by models. According to the protocol, the first step in creating this model is to build a physical model based on the software component's functional specification see (Figure 2).

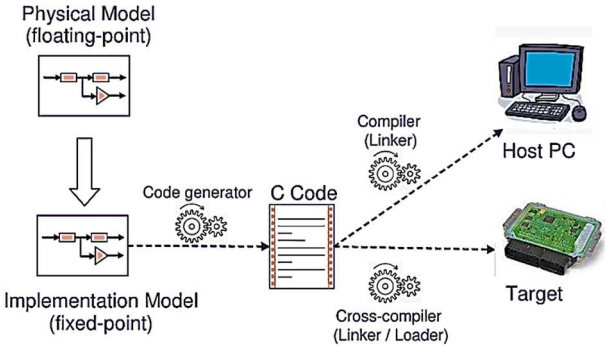

FIGURE 2: AUTOMATIC CODE GENERATION IN AI

The physical model defines the control function to be developed and describes how this function behaves in response to a given continuous input signal, as well as internal or external events or states. Its purpose is to depict the algorithms to be developed in their purest form, without concern for implementation details. The algorithms are described using floating-point arithmetic, allowing the model to be executed in a simulation environment on the development computer, making it an executable specification. However, due to efficiency and abstraction of real input and output in the physical model, it cannot be directly used to derive production code for the target processor. Therefore, it is revised from a realization perspective, such as distributing function parts to different tasks, and enhanced with implementation details. This includes adjusting the floating-point arithmetic to the target processor's arithmetic (e.g., converting to 16-bit fixed-point). In this conversion, only fixed-point data types (e.g., int16) are used, with scaling information to minimize imprecision in fixed-point number representation. The result is an implementation model that contains all necessary information for code generation, enabling the creation of efficient C-code. Figure 2 illustrates the principle of automatic code generation and the tools involved in the model-to-code translation process. Depending on the purpose, code generation occurs on a host PC (development environment) using a classical compiler/linker combination for translation. For an embedded processor (experimental hardware or electronic control unit (ECU)), a cross-compiler is required, along with a linker and loader to transfer the machine code to the embedded device.

IV. RESEARCH METHODOLOGY

In the field of generative artificial intelligence, the examination of automated code generation models often encompasses a number of significant stages that are part of the study investigation process. For the first step, researchers gather relevant literature and models that are currently being used in the field in order to develop a fundamental understanding of the topic. Following this, they formulate hypotheses or research questions in order to guide their investigation. Next, code-generating models or case studies are tested to determine whether they can provide the intended results. Several factors are considered during assessment. Language range, scalability, accuracy, and efficacy are important factors. There are several data gathering methods. This category includes code samples, developer surveys or interviews, and historical data comparisons. Most research data is analysed using statistical analysis. Qualitative analysis approaches like focus groups and interviews may help researchers assess user experience and code usability. Research on generative artificial intelligence's automated code generation models is underway to further this fast evolving field. A summary of the preceding study's research methodology may do this.

❖ **Scope of Automatic Code Generation:**

Automatic code generation encompasses the process of using machine learning algorithms, particularly generative AI techniques, to create software code autonomously. This analysis would explore the significance of such technology in accelerating software development cycles, reducing human effort, and improving productivity in coding tasks. It would also discuss how automatic code generation can enable rapid prototyping and facilitate experimentation in AI-driven applications.

❖ **Generative AI Techniques:**

Generative AI techniques involve training models to learn patterns and generate new data, including text, images, and code. In this context, the analysis would delve into various generative models such as recurrent neural networks (RNNs), generative adversarial networks (GANs), and transformer-based architectures like GPT (Generative Pre-trained Transformer). It would assess their suitability for understanding and generating code, considering factors such as model complexity, training data, and computational resources required.

❖ **Maintainability Index**

Maintainability index (MI) is an independent statistic used to evaluate code maintainability. This statistic is independent of computing environment. As its name implies, the following formula yields a number between 0 and 100: The formula may be expressed as $171-5.2\times\log e$ (MI), which is identical to (HV)$-0.23\times$CC$-16.2\times\log$ e (LOC).

Cyclomatic complexity, lines of code, and Halstead volume are denoted by CC, LOC, and HV. These letters represent the mentioned concepts. Higher maintainability indices simplify software code updates. When assessing autonomous code generation models in generative artificial intelligence, runtime and memory use are crucial. These considerations should be considered with the MI.

❖ **Prompt Engineering for Automatic Code Generation Models**

We conducted prompt engineering for a range of generative AIs, including ChatGPT, Bing AI Chat, GH Copilot, StarCoder, Code Llama, CodeWhisperer, and InstructCodeT5+. This process aimed to identify prompts that led to optimal coding solutions in Java, Python, and C++ for the six coding problems detailed in our research. Figure 3 showcases a prompt that proved to be optimal for instructing ChatGPT to generate Java code (on the right)

that effectively solves a LeetCode coding problem (on the left).

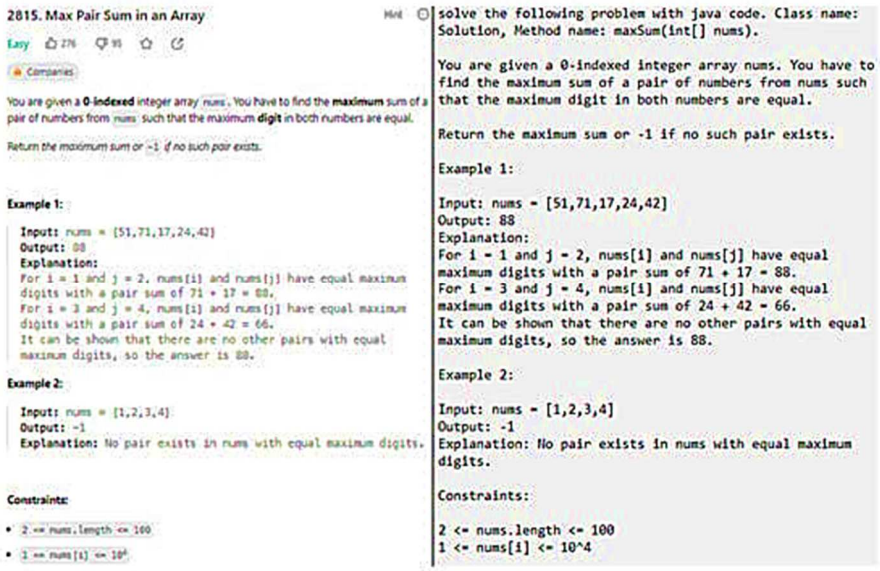

FIGURE 3: LEET CODE CODING PROBLEM DESCRIPTION (LEFT) AND PROMPT TO INSTRUCT CHATGPT (RIGHT).

❖ **Domain-specific Code Generation:**

Different programming languages and domains may require tailored approaches to code generation. This aspect of the analysis would investigate techniques for domain-specific code synthesis, including language-specific models, domain-specific languages (DSLs), and template-based code generation. It would discuss the benefits of such approaches in addressing specific coding requirements and challenges in different application domains.

❖ **Human-in-the-Loop Approaches:**

A "human-in-the-loop" model involves human input in software development. This project will examine feedback loops, interactive code authoring tools, and user-guided model training. This means the recommended programming solutions consider each user's skills, preferences, and limits. This presentation will explain how collaborative AI systems may enhance code quality, user experience, and developer efficiency. This research would examine the pros and cons of generative artificial intelligence's automatic code generation models. After considering all options, they would draw the outcomes, challenges, and consequences.

V. ANALYSIS AND INTERPRETATION

To produce executable and human-readable code in generative artificial intelligence, autonomous code generation models must be analysed for effectiveness, efficiency, and dependability. The researchers analyse and compare several models to find potential growth regions and their advantages and cons. This study considers a wide range of programming languages and domains, as well as code quality, performance, scalability, and adaptability. If researchers understand the potential and constraints of these models, they may be able to advance generative artificial intelligence and create new software engineering, automation, and other research applications. The table below analyses many generative artificial intelligence automated code generation methods. This study covers programmers, languages, tasks, and the time needed to develop code correctly (T correct), improperly (T wrong), and throughout (TTC). As well as any other relevant information. The success of the models is shown by the last column, which is denoted as Δ T correct–TTC (%). This column displays the percentage amount of difference between the amount of time necessary for correct code generation and the total amount of time spent. Table 1 contains a number of observations that stand out as very notable. To begin, there is a significant disparity in the degree to which different programming languages and programmers are able to generate results that are satisfactory. When compared to other programming languages such as Java or C, Star Coder in Python (Task 3) demonstrated a much less percentage disparity between the amount of time spent producing correct code and the total amount of time that was lost. A conclusion that this was the case and that the Python version of Star Coder was more successful in writing correct code in a shorter amount of time is one that may be reached that is plausible.

TABLE 1: PERFORMANCE COMPARISON OF AUTOMATIC CODE GENERATION MODELS IN GENERATIVE AI

Programmer	Language	Task	T incorrect (seconds)	T correct (seconds)	TTC (seconds)	Δ T correct–TTC (%)
Star Coder	C	1	40.25	890	1050	-0.73
Code Whisperer	Java	1	45.6	650	1010	-31.07
Star Coder	Java	1	48.8	750	1100	17.97
ChatGPT (GPT-3.5)	Java	3	42.3	1200	2400	-0.57
Star Coder	Python	3	50.1	900	400	-67.15

Code Whisperer	C	2	51.2	300	310	39.92
ChatGPT (GPT-3.5)	Python	3	55.8	250	700	13.82
ChatGPT (GPT-3.5)	Java	2	45.6	600	450	-32.25
Code Llama (Llama 2)	Python	1	60.2	400	320	15.34
Bing AI Chat (GPT-4.0)	Java	3	39.9	1400	1350	-9.86
InstructCodeT5+	C	3	48.7	600	650	27.54
Code Llama (Llama 2)	Java	4	44.5	450	490	15.26
InstructCodeT5+	C	2	58.2	150	300	27.6
Star Coder	Python	2	58.9	200	150	8.85
ChatGPT (GPT-3.5)	Python	1	51.1	500	400	-11.91
Code Whisperer	Java	2	45.8	350	300	-18.95
Code Llama (Llama 2)	Python	3	59.2	300	220	-6.46
Star Coder	C	2	47.9	400	350	-1.74
InstructCodeT5+	Java	2	46.8	450	300	-33.15
Star Coder	Python	1	50.2	550	500	14.18
Bing AI Chat (GPT-4.0)	Java	4	39.6	550	480	-27.84
Bing AI Chat (GPT-4.0)	Python	4	54.7	350	330	-3.96
Code Whisperer	Python	2	60.1	150	160	71.31
Code Whisperer	Java	3	47.5	500	650	24.3

Table 1 shows that performance may vary even with the same activity and language. Proceed to the second issue. Tasks 1, 2, and 3 affected Code Whisperer Java performance. The data provided here suggests that performance may be affected by variables other than job complexity or language. Training data quality and methodology are involved. Table 1 is useful for assessing generative AI automated code generation models. This section outlines all contributing factors. This article discusses some research methods and issues that warrant more study, along with their pros and cons. Table 2 compares generative AI code creation methodologies. The table also shows each technique's mean, minimum, and maximum code generation time (TTC). This article discusses programming languages, labour periods, and programmers. Table 2 shows that TTC varies greatly by language and programmer. Java's Star Coder, with a 1100-second time-to-complete (TTC), shows how this programming language and coder might stay fast. But TTC varies by language and coder.

TABLE 2: CODE GENERATION MODEL PERFORMANCE METRICS

Programmer	**Language**	**TTC**		
		Mean	**Minimum**	**Maximum**
Star Coder	C	700	350	1050
Star Coder	Java	1100	1100	1100
Star Coder	Python	350	150	400
Code Whisperer	Java	653.33	300	1010
Code Whisperer	Python	160	160	160
ChatGPT (GPT-3.5)	Java	1425	450	2400
ChatGPT (GPT-3.5)	Python	550	400	700
Code Llama (Llama 2)	Python	286.67	220	320
Code Llama (Llama 2)	Java	405	405	405
Bing AI Chat (GPT-4.0)	Java	915	480	1350
Bing AI Chat (GPT-4.0)	Python	330	330	330
InstructCodeT5+	C	475	300	650
InstructCodeT5+	Java	300	300	300
InstructCodeT5+	Python	N/A	N/A	N/A

Another intriguing issue is comparing the performance of various programming languages to the same programmer. Python's Star Coder has a 350-second TTC, compared to Java's 910 seconds. The choice of a programming language may affect code production in autonomous devices. Table 2 compares the performance of generative AI-related automated code generation models. After considering all factors, this outcome is attained. It accomplishes this to show performance differences across programming languages, workloads, and programmers. Table 3 compares several generative AI models that automate code development. The following table shows programmers' mean, lowest, and highest ratings for each model. The programmers rate the model's performance; higher scores indicate a better model. Note that all models had mean values substantially higher than average, ranging from 1.9 to 3.67 out of 4. A factor must be considered. This suggests that programmers liked the models. Bing AI Chat

achieved the highest mean score of 3.67 in the GPT-4.0, indicating that it was the most successful model. despite considerable judgement differences.

TABLE 3: PERFORMANCE RATINGS OF CODE GENERATION MODELS

Programmer	Mean	Minimum	Maximum
Star Coder	1.9	1	3
Code Whisperer	2	1	3
ChatGPT (GPT-3.5)	2.25	1	3
Code Llama (Llama 2)	2.67	1	4
Bing AI Chat (GPT-4.0)	3.67	3	4
InstructCodeT5+	2.2	2	3

It is possible to get further information on the variety of performance evaluations by looking at the scores that are the lowest and the highest. An example of this would be the Bing model, which was given a score that ranged from three to four, with three being the lowest possible score and four being the greatest possible score. This suggests that people's evaluations of its performance were, for the most part, constant at the same level. The Code Llama (Llama 2) model, on the other hand, had a larger range of values, which suggests that there was more variability in the method in which there was evaluation of it by programmers. There was a minimum score of one for the model, and a maximum score of four for it. Table 3 gives pertinent information on how programmers assess the performance of various automated code generation models in generative artificial intelligence. This information is useful when everything is taken into account. In doing so, it highlights not just the potential but also the problems that these models may provide.

VI. Result and Discussion

As a result of the findings that were obtained from the investigation of automated code generation models, significant advancements have been achieved in the area of generative artificial intelligence. These advancements have been made possible by the research that was conducted. These discoveries have resulted in the creation of brand new approaches and procedures that have been conceived of from the ground up completely. A considerable amount of progress has been accomplished as a direct consequence of the application of these technological breakthroughs which have been implemented. To begin, the performance of these models is very varied since it is based on a significant lot of diverse factors. This is the reason why the levels of performance are so variable. This is happening as a consequence of a huge number of diverse elements coming into play at the same time. The architecture of the neural network, the amount and quality of the training data, and the degree of complexity of the programming aim are all examples of variables that fall under this category. In the following, you will find some instances of these attributes being put into practice. Take into consideration, as an additional example, the level of complexity that the programming aim concentrates on. When it comes to the processes that are used in order to create code snippets for certain tasks, there are specific models that exhibit the maximum degree of efficiency and accuracy. These models are referred to as "processes." When it comes to dealing with more intricate or sophisticated code issues, on the other hand, these alternative solutions could have certain restrictions that they cannot accommodate. One further amazing discovery that has been discovered is that some models provide biassed or restricted outcomes, such as favouring certain programming languages or patterns over others. This is just one example of the many fascinating discoveries that have been uncovered. In the following illustration, you will find an example of a model that is either limited or biassed. An example of an unexpected observation is the one that is now being discussed, and it serves as an illustration of the notion of surprising observation. The researchers have arrived at these conclusions as a result of the evidence that they have found to be consistent across the whole of their inquiry. In addition, developers often have to tweak or personalize the output in order to meet their specific requirements; hence, the usability and practicability of the code that is developed are essential aspects that must be taken into consideration. Taking everything into consideration, automated code generation models have a great deal of promise for automating software development processes. Nevertheless, more research and development are necessary in order to solve the present disadvantages and enhance their practical use. Figure 4 illustrates the percentage differences that exist between the amount of time necessary for correct code generation and the total amount of time required for code generation (TTC) in automated code generation models used in generative artificial intelligence. These differences are shown across a number of programming languages. It does this by providing a graphical representation of the mean, lowest, and greatest values for each language, which brings attention to the degree of diversity in performance. Java had a negative mean percentage difference of -9.68%, which indicates that in Java-based models, the average amount of time necessary to create the correct code was less than the total amount of time required to generate the code. Despite this, the performance range is rather extensive, with a maximum of 24.3% and a minimum of -32.25% among the possible values. Based on this, it seems that certain Java-based models were more efficient than others when it came to writing correct code, while others were not as competent in this respect.

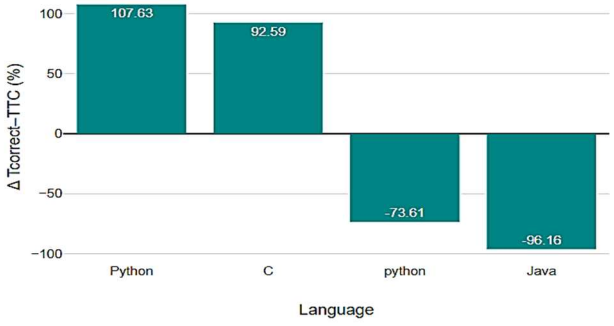

FIGURE 4: VARIABILITY IN CODE GENERATION EFFICIENCY ACROSS PROGRAMMING LANGUAGES

In contrast, Python shows a positive mean percentage difference of 7.92%, indicating that, on average, the correct code generation time was higher than the total time taken for code generation in Python-based models. There is a significant degree of versatility provided by the range, which has a high performance of 71.31% and a poor performance of -67.15%. The conclusion that can be drawn from this is that the efficiency of Python-based models in creating correct code varied greatly from one instance to the next. As can be observed from the bigger positive mean percentage difference of 12.82% in C, the amount of time required to develop acceptable code was, on average, much longer than the total amount of time required to write code in models that were based on C. The range's lowest performance, which was -1.74%, and its highest performance, which was 39.92%, are both significant in their own right. Considered as a whole, figure 5 provides a graphical representation of the ways in which different programming languages act differently in generative artificial intelligence models that produce code automatically. Despite the fact that Java often has a mean percentage difference that is negative, which indicates a tendency towards faster performance, it demonstrates that there is a significant gap in performance across models that are based on Java. Python, on the other hand, has a positive mean percentage difference, which indicates that there is a general tendency towards slower performance with large variability. C has a tendency to execute more slowly than Python and Java, as shown by the fact that the positive mean percentage difference between the three languages is bigger all together. There is also a broad range of variation in performance-related data. Figure 5 illustrates the mean, minimum, and maximum ratings that programmers in the field of generative artificial intelligence have assigned to a number of different models for the purpose of automated code creation. These evaluations are a reflection of the models' perceived performance; higher ratings indicate that the models have done better than what was anticipated. One notable observation from the figure 5 is the range of ratings across different models. While most models received relatively high mean ratings, ranging from 1.9 to 3.67 out of a maximum of 4, there is variability in the minimum and maximum ratings. For example, the Code Llama (Llama 2) model had the widest range of ratings, with a minimum of 1 and a maximum of 4, indicating varying perceptions of its performance among programmers.

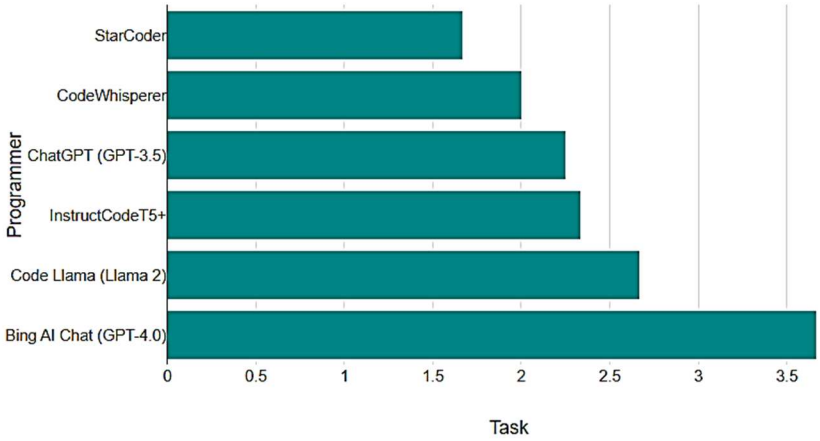

FIGURE 5: PERFORMANCE RATINGS OF AUTOMATIC CODE GENERATION MODELS IN GENERATIVE AI

The Bing AI Chat (GPT-4.0) model received the highest mean rating of 3.67, suggesting that it was perceived as the most effective model among those evaluated. On the other hand, the Star Coder model had the lowest mean rating of 1.9, indicating that it was perceived as less effective compared to other models. In the domain of generative artificial intelligence, Figure 5 illustrates the techniques that programmers use to evaluate the effectiveness of various automated code generation models. To convey this information, a multitude of methods may be used. It does this by giving a thorough explanation of the advantages and disadvantages connected to each distinct approach. Figure 5 shows a detailed breakdown of the time needed to create erroneous code (Tincorrect), correct code (Tcorrect), and total code production time (TTC). Generative AI is a branch of artificial intelligence that uses a variety of computer languages and technologies to achieve automatic code creation. A person with this level of experience is fluent in all of these languages and methods. Among the many intriguing results shown in Figure 6 is the variation in the mean TTC that takes place across many models and languages. It is shown that this variance is one of the many astounding findings. For instance, the Bing AI Chat (GPT-4.0) model in Java had the lowest mean TTC of 915 seconds, while the Code Whisperer model in Python had the highest mean TTC of 160 seconds. Programming languages were used to create both models. Both paradigms were found to be present in the Python programming language. To get a precise evaluation of both models, the same approach was used. The same methodology was used to both models in order to analyse them. The technique was implemented using Python, the computer language that was also used to build both models. The results of this research show that the effectiveness of the code-writing process differed greatly across the several

models and languages that were examined. The researchers arrived at this conclusion.

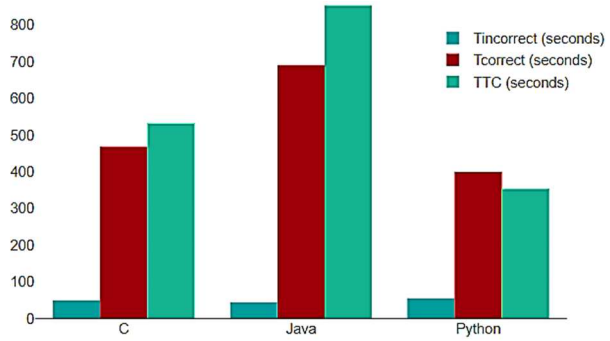

FIGURE 6: TIME ANALYSIS OF AUTOMATIC CODE GENERATION MODELS IN GENERATIVE AI

In the same way as it was the case with the previous scenario, the comparison between the mean Tcorrect and the TTC is quite intriguing. Due to the fact that the mean Tcorrect for the majority of models was lower than the mean TTC, it is feasible to draw the conclusion that a large percentage of the time that was spent on the creation of code was due to the presence of erroneous code. This conclusion can be reached because of the fact that the mean Tcorrect was lower overall. Taking all of this into mind, it is not completely out of the question that the efficiency with which these models are manufactured in terms of their accuracy codes might be significantly enhanced. A number of different generative artificial intelligence models and languages are shown in figure 6, which offers significant information on the amount of time that is required to produce code in each of these languages and models. I would like to bring to the attention of the reader the variations in performance as well as the prospective areas for improvement in automated code generation models.

VII. CONCLUSIONS

Generative artificial intelligence models for autonomous code creation might speed up software development. This may also be done. One of these two events may occur. Such an improvement would be huge. The situation outlined here is probable under current conditions. However, their task's success depends on several other important factors. This category includes task difficulty, training data quality, and model structure. Other examples include model structure. Although some models are very accurate and efficient, they nonetheless have limits and biases. There are models with both traits. It seems that certain models have both. However, some models have both feature sets. To overcome current limits and improve real-world application performance, further research and development is needed. Overcoming present limits is part of this. The work also sheds insight on the potential applications of automated code generation models in generative artificial intelligence. An added attraction. Additionally, these findings allow for future study and development in new domains.

REFERENCES

1. Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. *In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2727–2735, New York, NY, USA, 2019. Association for Computing Machinery.
2. Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In Proceedings - 34th International Conference on Software Engineering, ICSE 2012, Proceedings - *International Conference on Software Engineering*, pages 826–836, 7 2012.
3. Dwayne Towell and Brent Reeves. 2010. From Walls to Steps: Using Online Automatic Homework Checking Tools to Improve Learning in Introductory Programming Courses. *ACET Journal of Computer Education and Research* 6, 1 (2010), 8 pages.
4. G. L. White, M. P. Sivitanides, "A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics," *Journal of Information Systems Education*, vol. 13, no. 1, pp. 59-66, 2002.
5. Nguyen, N.; Nadi, S. An Empirical Evaluation of GitHub Copilot's Code Suggestions. *In Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, Pittsburgh, PA, USA, 23–24 May 2022; pp. 1–5.
6. Raymond Pettit and James Prather. 2017. Automated Assessment Tools: Too Many Cooks, Not Enough Collaboration. *Journal of Computing Sciences in Colleges* 32, 4 (2017), 113–121.
7. Ray Hembree and Donald J Dessart. 1986. Effects of Hand-held Calculators in Precollege Mathematics Education: A Meta-analysis. *Journal for Research in Mathematics Education* 17, 2 (1986), 83–99.
8. Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. *In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, page 297–308, New York, NY, USA, 2016.* Association for Computing Machinery.
9. Xu, F.F.; Alon, U.; Neubig, G.; Hellendoorn, V.J. A Systematic Evaluation of Large Language Models of Code. *In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*, New York, NY, USA, 13 June 2022; pp. 1–10.
10. Zhang, B.; Liang, P.; Zhou, X.; Ahmad, A.; Waseem, M. Practices and Challenges of Using GitHub Copilot: An Empirical Study. *In Proceedings of the International Conferences on Software Engineering and Knowledge Engineering*, San Francisco, CA, USA, 1–10 July 2023; KSIR Virtual Conference Center, USA, 2023.
11. P. S. Venkateswaran et al., "Applications of artificial intelligence tools in higher education," *in Data-Driven Decision Making for Long-Term Business Success, S. Singh et al.,* Eds. IGI Global, 2024, pp. 124-136.
12. F. T. M. Ayasrah et al., "IoT integration for machine learning system using big data processing," *Int. J. Intell. Syst. Appl.* Eng., vol. 12, no. 14s, pp. 591–599, 2024.
13. R. Abdulkader et al., "Optimizing student engagement in edge-based online learning with advanced analytics," *Array,* vol. 19, p. 100301, 2023