A Comparative Review of AI Techniques for Automated Code Generation in Software Development: Advancements, Challenges, and Future Directions

Ayman Odeh [1], Nada Odeh [1], Abdul Salam Mohammed [2]

[1] *Department of Software Engineering, Colleg of Engineering, Al Ain University, Al Ain, UAE*
[2] *Skyline University College, Dubai, UAE*

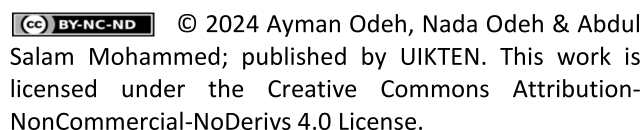
Abstract **–Artificial Intelligence (AI), as one of the most important fields of computer science, plays a significant role in the software development life cycle process, especially in the implementation phase, where developers require considerable effort to convert software requirements and design into code. Automated Code Generation (ACG) using AI can help in this phase. Automating the code generation process is becoming increasingly popular as a solution to address various software development challenges and increase productivity. In this work, we provide a comprehensive review and discussion of traditional and AI techniques used for ACG, their challenges, and limitations. By analysing a selection of related studies, we will identify all AI methods and algorithms used for ACG, extracting the evaluation metrics and criteria such as Accuracy, Efficiency, Scalability, Correctness, Generalization, and more. These criteria will be used to perform a comparative result for AI methods used for ACG, exploring their applications, strengths, weaknesses, performance, and future applications.**

Keywords **–Artificial intelligence, automated code generation, deep learning, evolutionary algorithms, machine learning, natural language processing.**

DOI: 10.18421/TEM131-76
https://doi.org/10.18421/TEM131-76

Corresponding author: Ayman Odeh,
Department of Software Engineering, Colleg of Engineering, Al Ain University, Al Ain, UAE
Email: ayman.odeh@aau.ac.ae

Received: 03 October 2023.
Revised: 17 January 2024.
Accepted: 23 January 2024.
Published: 27 February 2024.


© 2024 Ayman Odeh, Nada Odeh & Abdul Salam Mohammed; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at https://www.temjournal.com/

1. Introduction

AI is rapidly changing the software development process, it has the potential to significantly improve the software engineering process [1]. Software development is a complex and demanding process that involves analysis and coding phases [2], [3]. However, it is also known to be expensive, especially in environments that adhere to procedures, standards, and team structures [4]. Automated code generation (ACG) mentioned in [5] is becoming increasingly important to software development, especially through the use of machine learning (ML) and artificial intelligence (AI). It focuses on using a transformer-based ML model to generate frontend component code for angular, using behavior-driven development (BDD) test specifications as input. ACG techniques aim to automate repetitive and time-consuming tasks, allowing developers to focus on higher-level design and problem-solving. In recent years, researchers have explored the utilization of artificial intelligence (AI), particularly deep learning (DL) techniques, for ACG due to significant advancements in this field. The use of AIT Techniques (AIT) in ACG has revolutionized the software development field. Machine learning (ML) and DL algorithms have been applied to automate various aspects of the code generation process [6], [7]. Researchers have conducted surveys and systematic reviews to investigate the effectiveness of different approaches, such as natural language processing (NLP) and source code analysis, in generating code automatically [8], [9]. Developers have even been assisted in writing code more efficiently by AI-powered coding companions like those introduced by Amazon [10]. Also many of application of AI in code generation has been explored in different domains such as web development, mobile applications, and industrial automation [11], [12], [13].

This progress has led to the development of models and tools capable of generating code from natural language (NL) descriptions [14], sketches, and other input forms [15], [16]. The significance of these AIT is evident in the acceleration of software development and the reduction of programming efforts [17], [18]. However, challenges remain, including evaluating the performance of AI-based code generators and ensuring the quality of the generated code [19], [20].

1.1. Problem Statement and Significance

The problem at hand is to evaluate and compare different AIT for ACG in software development, where AI can improve all phases of the software development life cycle [21]. This research aims to understand the strengths and weaknesses of various AITs, assess their suitability for different code generation tasks, and identify areas for improvement.

By providing a comparative review of AIT, this paper aims to assist researchers and practitioners in selecting appropriate techniques for ACG, ultimately enhancing the efficiency and quality of software development processes.

1.2. Aims and Motivation

The aim of this paper is to provide a comprehensive and comparative analysis of AIT for ACG in software development; and analyze advancements, challenges, and future directions in this field. By comparing different AIT, including their strengths, weaknesses, and real-world applications, the paper aims to facilitate informed decision-making and promote the adoption of effective AI-based code generation practices. The motivation behind conducting a comparative review of AIT for ACG in software development stems from several key factors as shown in Table 1.

Table 1. Motivation factors and achieved objects

No	Motivation factors	Achieved objects
1	**Advancements in AI**	To evaluate the effectiveness and identify strengths and weaknesses of various AITs for ACG
2	**Increasing Demand for Automated Code Generation**	To pinpoint the most effective AIT and assess their suitability across diverse development scenarios.
3	**Varied AIT in Code Generation**	Understanding the trade-offs and selecting the most appropriate technique based on factors such as code quality, scalability, and efficiency
4	**Addressing Challenges and Limitations**	Identifying these challenges and assessing the extent to which different AIT mitigate or exacerbate them. By understanding the limitations, researchers can focus on improving these techniques and practitioners can make informed decisions about their implementation
5	**Future Directions and Research Opportunities**	Identify gaps in the existing approaches and propose future research directions.

1.3. The Paper Structure

The paper is organized as follows: Section 2 provides literature review and discussion of traditional and AI approaches to code generation and their limitations. In Section 3, we present advancements in AIT for ACG. Section 4 discusses the challenges and limitations of AI-based code generation. Section 5 provides a comparative analysis of these AITs, emphasizing their strengths and weaknesses. Section 6 presents future directions and research opportunities. Finally, Section 7 concludes the paper, summarizing the key findings and providing recommendations for researchers and practitioners.

2. Literature Review

This section summarizes the wide range of methodologies and technologies that have significantly aided the advancement of ACG. Our investigation is divided into six critical approaches, each representing a distinct viewpoint and methodology in the field of code generation.

2.1. Traditional Approaches

Traditional automated code generation methods, predating ML and DL algorithms, rely on rule-based (RB) systems, template-based (TB) methods, and other conventional programming techniques [22]. RB code generation involves predefined rules or patterns to transform high-level specifications into executable code, particularly useful in domains with repetitive and well-defined code patterns.

TB code generation utilizes pre-defined code fragments with placeholders, allowing developers to input specific values or logic. These templates are instantiated by the code generation system, commonly used in frameworks or specialized code generators for rapid code generation in specific languages or domains.

Additionally, traditional methods incorporate domain-specific languages (DSLs) or modeling languages [20], providing expressive, domain-specific syntax for articulating code generation requirements. The code generation system translates these high-level specifications into actual code in the target programming language. These traditional techniques automate the code generation process through predefined rules, templates, and domain-specific abstractions [22]. Despite their effectiveness, they often require manual efforts in rule or template definition and may have limitations in handling complex or evolving code generation tasks.

2.2. Rule-Based Systems

RB code generation relies on predefined rules and patterns to generate code from high-level specifications or natural language descriptions. These systems use expert knowledge encoded in rules to map input specifications to code structures. RB systems have been widely used in industrial automation and specific application domains where the generation process can be well-defined and RB [13], [23]. They offer interpretability and control over the generated code but may require extensive rule engineering and may not handle complex or ambiguous specifications effectively. RB systems use a set of rules to generate code. The rules are typically expressed in a declarative language, such as Prolog [24].

2.3. Machine Learning (ML)

ML techniques have been extensively explored for code generation tasks. ML-based code generation approaches often involve training models on large code repositories to learn patterns, relationships, and common coding practices. These models can generate code based on the learned knowledge. ML techniques such as statistical language models, recurrent neural networks (RNNs), and transformers have been employed for tasks like auto-completion, code summarization, and code generation [25], [2], [11]. ML-based approaches can capture complex patterns in the data but may struggle with rare or unseen coding scenarios and require substantial amounts of training data.

For example, A ML-powered service from Amazon, CodeWhisperer [26], provides code recommendations for developers based on their natural comments.

2.4. Natural Language Processing (NLP)

NLP can be used to generate code from NL descriptions [27], [28]. NLP techniques such as parsing, neural language models, sequence-to-sequence models [11], semantic analysis, and machine translation can be used for this purpose. One more study [29] focuses on integrating external knowledge sources to enhance NLP-based code generation. The research [27] uses DL techniques for code generation from NL description. It investigates neural network architectures, such as sequence-to-sequence models, to capture the intricate relationships between NL and code. It emphasizes the ability of DL models to handle long-range dependencies in NL descriptions, leading to improved code generation performance [27].

2.5. Deep Learning (DL)

DL models, such as CodeGRU and deep transfer learning, have been developed to model and generate source code [30], [31]. Recurrent neural networks (RNNs)[32] can capture sequential dependencies in code and generate code snippets or complete functions. Transformers (e.g., GPT [33], [34], BERT [35]), have been widely used for code generation [27], [30], [31], allowing the model to attend to relevant code contexts and generate code with improved context awareness [7]. Graph neural networks (GNNs) [36] can handle code represented as graphs and capture relationships between code entities [15]. DL techniques excel in capturing complex patterns and generating code with improved accuracy [7]. However, they require large amounts of labeled training data, substantial computational resources, and may suffer from a lack of interpretability.

2.6. Evolutionary Algorithms (EAs)

EAs are a type of ML algorithm that can be used to generate code by iteratively mutating and evolving existing code. EAs work by starting with a population of randomly generated code, and then iteratively selecting the best performing individuals from the population and mutating them to create new individuals. This process is repeated until a satisfactory solution is found. EAs benefits: they can be used to: generate code that is efficient and effective [37], easy to maintain [38], for different programming languages, platforms, and applications [37].

The research provided in [39], proposes a new approach to software development that uses artificial agents to automate the code generation process. Another study [21] explores the integration of AI into the Software Development Life Cycle (SDLC). The authors posit that AI can automate numerous tasks within the SDLC, including requirements gathering, analysis, and validation. Such integration has the potential to yield substantial productivity enhancements and elevate software quality.

3. The Advancements in AIT for ACG

The advancement of AIT has led to a significant rise in the popularity of ACG in recent years [21]. Various AITs have been employed for code generation, including RB systems, ML, DL, NLP, and EAs.

3.1. The Recent Advancements

This section provides the recent and the most important advancements in AI for ACG in various domains, as well as the following AI applications used for ACG as shown in Table 2.

Table 2. AI Applications

No	AI Application	Description
1	CodeGRU	A context-aware Deep Learning (DL) model with gated recurrent unit (GRU) architecture for source code modeling [30].
2	Amazon CodeWhisperer	An Machine Learning (ML)-powered coding companion developed by Amazon that assists developers in writing code more efficiently and effectively [26].
3	BERTGen	A multi-task generation model based on BERT (Bidirectional Encoder Representations from Transformers) for code generation tasks [35].
4	GPT-3	A state-of-the-art language model developed by OpenAI [41] that can be used for various Natural Language Processing (NLP) tasks [42], including code generation.
5	DeepCoder	A neural network with leaky ReLU achieves the best performance when compared to other approaches[45].
6	DL Code Completion	A DL-based code completion approach that uses language models to predict the next code token given a partial code sequence [43].
7	DL Code Editors (e.g., GitHub's Copilot) [41]	Code editors powered by DL models that provide intelligent code completion and generation suggestions to developers, and can even write entire programs [43], [44]
8	CodeGAN	A model utilizing neural networks and NLP techniques to generate code from high-level descriptions or code snippets [6].
9	AlphaCode	A DL model achieving human-level performance on the Codeforces platform[46].
10	Tree-Structured Architectures	Approaches using tree-based representations of code syntax to guide code generation and enhance the structural coherence of generated code [19].
11	RB Code Generation	Techniques using Rule-Based (RB) systems to generate code based on predefined rules and patterns[23], [47].
12	EAs for Code Generation	Genetic programming and Evolutionary Algorithms (EAs) applied to code generation tasks, optimizing code generation through evolutionary processes [38], [48], [49].
13	TB Code Generation	Approaches using templates and patterns to generate code based on predefined structures and rules [50], [51], [52].
14	Frameworks (e.g., Tensorflow, PyTorch)	Powerful tools and APIs for building and deploying code generation models, enabling researchers and developers to use cutting-edge DL techniques [2], [25]
15	Google Cloud AutoML Code [50].	A service that uses ML to generate code for various programming languages and platforms [53].

3.2. The Benefits and Improvements Achieved Through AI-based Code Generation

1. **Increased productivity and efficiency**: AI-based code generation automates repetitive and time-consuming tasks in software development, allowing developers to focus on higher-level tasks. It accelerates the coding process by generating code snippets, templates, or even complete programs, reducing the overall development time and effort [18], [25].

2. **Enhanced code quality:** AI models can analyze large codebases, identify patterns, and learn best coding practices from existing high-quality code.

This enables them to generate code that adheres to industry standards, follows coding conventions, and incorporates good software engineering principles. The generated code is less prone to errors and exhibits improved readability, maintainability, and modularity [25], [2].

3. **Enabling code generation from alternative representations:** AI models can generate code from different representations, such as images, diagrams, or sketches. This allows developers to express their ideas visually or graphically and automatically convert them into executable code. It promotes low-code or no-code development and empowers individuals with limited coding skills to create functional applications[54], [55].

4. **Code completion and autocompletion:** AI models trained on vast code repositories can provide intelligent code completion suggestions based on the context. They can predict the next lines of code, recommend suitable function calls, suggest variable names, and provide helpful documentation.

This feature speeds up the coding process and reduces the likelihood of syntactic or logical errors [25], [43].

5. **Support for code refactoring:** AI models can assist in refactoring code by automatically suggesting improvements or generating refactored code snippets. This helps developers improve the structure, organization, and performance of existing codebases [2].

6. **Transfer learning and knowledge sharing:** AI models trained on large codebases can capture the knowledge and expertise embedded within the code [31].

7. **Continuous learning and improvement:** AI models can be continuously trained on new code repositories, incorporating the latest coding practices and trends [6], [56].

8. **Bridging the gap between NL and code**: AI models can understand NL of software requirements or functionalities and generate corresponding code [27], [10].

Table 3provides some benefits, impact, and lessons learned from the applications of AIT ACG.

Table 3. Impact, benefits, and lessons learned from ATIs applications for ACG

Application	Benefits	Impact	Lessons Learned
Amazon CodeWhisperer [26]	- Context-aware code completion and bug detection	- Improved coding efficiency	- AI-powered coding companions can enhance developer productivity
Automatic HTML Code Generation [57]][9]	- Faster front-end development	- Reduced manual coding efforts	- AIT can automate repetitive and time-consuming tasks in web development
Mobile Application Development [55]	- Simplified mobile app development	- Reduced programming efforts	- AI-based code generation from sketches can facilitate rapid prototyping and development of mobile applications
Function Block Applications [37]	- Automatic generation of function block applications	- Faster and more efficient development in automation	- Evolutionary algorithms can optimize industrial automation systems
Source Code Modeling and Generation [43]	- ACG based on learned patterns	- Improved code quality and consistency	- DL models can capture complex patterns and structures in source code
Natural Language to Code Generation [58]	- Translation of natural language descriptions into code.	- Enables non-programmers to express intentions in code	- NLP techniques can bridge the gap between human language and programming languages.

4. The Challenges, Limitations, and Ethical Considerations

The utilization of ACG with AITs has garnered substantial interest recently owing to its capacity to enhance productivity and efficiency in software development. Nevertheless, there are various challenges and constraints that necessitate careful consideration. This section delves into some of these challenges and limitations.

1. **Lack of Sufficient Training Data**: One major challenge in AI-based code generation is the availability of high-quality training data. Training ML models require large and diverse datasets that accurately represent the target problem domain. However, obtaining such datasets for code generation tasks can be difficult due to the proprietary nature of codebases or limited access to labeled code examples [25], [2].

2. **Lack of Contextual Understanding**: AI models may struggle with understanding the context and requirements of code generation tasks, especially when dealing with complex or domain-specific scenarios [1], [54]. It can be challenging to capture the nuances of programming languages, frameworks, and libraries, leading to suboptimal code generation.
3. **Limited Training Data:** Training data availability can be a significant challenge in code generation tasks. Generating high-quality and diverse training datasets that cover various programming languages, frameworks, and coding styles is often difficult [2], [43]. Limited training data can impact the performance and generalization ability of AI models.
4. **Difficulty in Handling Ambiguities:** Code generation tasks often involve dealing with ambiguous or incomplete specifications, making it challenging for AI models to generate accurate and desired code [59], [34]. Ambiguities in natural language descriptions or incomplete requirements can result in code that does not meet the intended functionality.
5. **Scalability and Performance**: Scaling AI models for large-scale code generation can be computationally expensive and time-consuming[33], [10]. Generating complex codebases or working with massive code repositories can pose challenges in terms of memory, computation, and efficiency.
6. **Overfitting and Generalization**: AI models trained for code generation tasks are prone to overfitting, where they memorize specific patterns from the training data but struggle to generalize to unseen code examples. This limitation can lead to the production of code that lacks robustness and fails to handle edge cases or adapt to different scenarios. Achieving a balance between capturing common patterns and promoting generalization is a significant challenge in AI-based code generation [25], [7], [30] [55], [27].
7. **Maintenance and Adaptation:** Code generation models need to adapt to changing programming languages, libraries, and frameworks [38], [13]. Maintaining and updating these models to accommodate new language features and coding practices can be a time-consuming and resource-intensive process.
8. **Balancing Flexibility and Guided Generation:** Striking a balance between generating code that meets specific requirements while allowing flexibility for developers to customize and modify the generated code can be challenging [19], [50]. AI models need to provide options for customization without overwhelming the developer with an excessive number of choices.
9. **Trust and Safety:** As AI models are used for automated code generation, ensuring the trustworthiness and safety of the generated code becomes crucial [22], [51]. Issues such as bias, security vulnerabilities, and unintended consequences in the generated code need to be addressed.
10. **Adoption and Acceptance:** Widespread adoption and acceptance of AIT for ACG may face resistance and skepticism from developers and industry stakeholders [60], [4]. Building trust, demonstrating the value, and addressing concerns around job displacement and loss of control are important factors in the successful adoption of AIT in code generation.
11. **Code Complexity and Variability**: Codebases can be highly complex and vary significantly across different projects and programming languages. AIT for code generation often struggle with capturing and understanding the nuances and intricacies of code syntax, semantics, and idiomatic patterns. This makes it challenging to generate accurate and high-quality code that aligns with the desired functionality and style [25], [2], [7].
12. **Difficulty in Capturing Context and Intent**: Understanding the context and intent of the code generation task is crucial for producing correct and meaningful code. However, AI models may struggle to capture the complete context and accurately interpret the developer's intent from limited information, such as code snippets or natural language descriptions. Ambiguities in specifications or lack of explicit requirements further complicate the code generation process [43], [27], [28].
13. **Limited Support for Domain-Specific Languages and Libraries**: AIT for code generation often focus on popular programming languages and libraries, such as Python or TensorFlow. However, many software development projects utilize domain-specific languages or libraries that are less widely studied or have limited available training data. Adapting AI models to support such specialized domains can be challenging due to the lack of resources and specialized knowledge required [25], [2].
14. **Debugging and Maintenance Challenges**: Generated code produced by AI models may contain bugs, logical errors, or suboptimal performance, which can be challenging to identify and fix. Debugging and maintaining code generated by AI systems can be more complex than traditional human-written code. This challenge raises concerns about the reliability and maintainability of code generated through AIT [18], [26], [54].

4.1. Ethical Considerations

The use of AIT in ACG raises ethical and legal concerns, particularly when it comes to generating code that is used in safety-critical systems or handles sensitive data. Ensuring the generated code adheres to security, privacy, and ethical guidelines poses significant challenges and requires careful validation and verification processes [18], [26]. AI-generated code presents various ethical considerations, biases, and potential risks that need to be addressed to ensure responsible and safe use. Here are some key points to consider:

1. **Bias and Fairness**: AI models trained on biased or incomplete data can lead to biased code generation. If the training data primarily represents a specific demographic or excludes certain groups, the generated code may reflect those biases, perpetuating inequality and discrimination. It is essential to address bias during the training process and incorporate diverse and representative datasets [25], [11].
2. **Reliability and Accountability**: AI-generated code may contain errors or produce unintended consequences. It is crucial to verify and test the generated code thoroughly to ensure its correctness, robustness, and safety. Developers and users should be aware of the limitations of AI-generated code and take responsibility for its outcomes [18].
3. **Privacy and Security**: AI models used for code generation may process sensitive or proprietary information. Safeguarding data privacy and ensuring secure code generation are paramount. Developers must implement strict security measures to protect confidential information and prevent malicious use of AI-generated code [26].
4. **Transparency and Explainability**: AI models used for code generation often operate as black boxes, making it challenging to understand the underlying decision-making process [61]. Enhancing transparency and explainability in AI systems is crucial to build trust and enable effective debugging, auditing, and compliance with legal and ethical standards [2], [43].
5. **Intellectual Property and Copyright**: AI-generated code may raise concerns regarding intellectual property and copyright. Developers need to ensure that generated code adheres to legal and licensing requirements and does not violate intellectual property rights [54].
6. **Unemployment and Job Displacement**: The automation of software development through AI-generated code has the potential to impact employment in the software engineering field. While AI can augment developers' capabilities, there is a risk of job displacement. Efforts should be made to reskill and upskill individuals to adapt to the changing landscape and mitigate the negative impact on employment [1].
7. **Human Oversight and Control**: AI-generated code should not replace human decision-making entirely. It is crucial to maintain human oversight and control over the generated code to ensure its alignment with ethical and legal standards. Human intervention is necessary to review, validate, and modify the generated code as needed [43], [7].

Table 4 provides a summary of main ethical considerations, biases, and potential risks [62] related to using AIT for ACG. They should be managed carefully to ensure the responsible and ethical use of ACG in software development.

Table 4. Ethical considerations, biases, and potential risks

Ethical Considerations	Biases	Potential Risks
Bias and Fairness	Bias in training data	Errors and unintended consequences
Reliability and Accountability	Incomplete or biased datasets	Lack of code correctness and robustness
Privacy and Security	Exposing sensitive information	Misuse of confidential data
Transparency and Explainability	Lack of transparency and explainability	Difficulty in understanding decision-making process
Intellectual Property and Copyright	Violation of IP and copyright	Legal consequences and infringements
Unemployment and Job Displacement	Impact on employment in the software engineering field	Job displacement and unemployment
Human Oversight and Control	Lack of human intervention and review	Violation of ethical and legal standards

5. Comparative Analysis of AI T for ACG

In this key section, we will compare and contrast different AI techniques used for generating code automatically. We will look at how they work, how they are measured, and their strengths and weaknesses.

5.1. Methodology

In the Methodology section our approach involves a process that aims to understand how AITs are used for ACG. Initially we carefully gathered data, from the research articles mentioned in the reference section ensuring a foundation for our analysis.

Then we explored a range of AI methods applied in ACG examining how they are implemented in contexts. To provide insights we identified the AI algorithms used in each method and mapped out the intricate technological landscape. Moreover we meticulously identified the criteria used to evaluate each method emphasizing their effectiveness and real world applicability.

During our analysis we critically evaluated both strengths and weaknesses of each method to offer a perspective on their performance. This comprehensive approach allowed us to gain nuanced insights into the evolving field of AIT for ACG and laid the groundwork, for discussions and valuable conclusions. Figure 1 depicts the proposed structure.

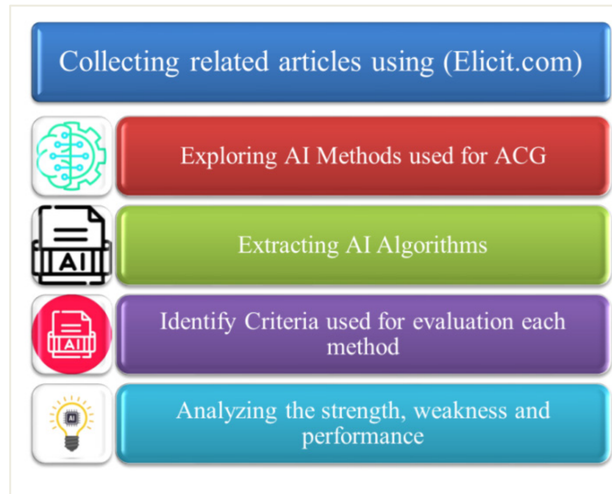


Figure. 1. The general methodology steps

5.2. Evaluation Criteria

To evaluate the strengths, weaknesses, and performance of different AITs for ACG, several criteria can be considered. In the Table 5, we provide the most common and important criteria.

Table 5. The evaluation criteria of AI methods used for ACG

No	Criteria	Description
1	**Accuracy**	Measure of how well the technique performs in generating correct and functional code [18], [25], [2], [43], [63], [39], [21].
2	**Efficiency**	The efficiency of the generated code is evaluated based on performance metrics such as execution time, memory usage, and computational complexity [40], [63].
3	**Scalability**	Evaluation of how well the technique can handle large-scale codebases and datasets [25], [2], [11].
4	**Generalization**	Ability of the technique to generalize and perform well on unseen or diverse code examples [25], [11], [1].
5	**Interpretability**	Degree to which the technique provides understandable and interpretable results [1], [54], [7].
6	**Robustness**	Resilience of the technique to handle noisy or incomplete input code [2], [7], [10]. Robustness evaluation helps ensure the reliability and adaptability of the ACG system.
7	**Adaptability**	Capability of the technique to adapt to changing code requirements or contexts [20], [21], [40].
8	**Training Data Requirements**	Assessment of the amount and quality of labeled training data needed for the technique [2], [54].
9	**Handling Rare Patterns**	Evaluation of the technique's ability to handle rare or uncommon code patterns [9].
10	**Language Support**	Assessment of the technique's applicability and effectiveness across different programming languages [11], [59], [27].
11	**Correctness**	The generated code should be correct and function as intended. It should accurately implement the desired functionality without introducing errors or bugs. Evaluating correctness involves comparing the generated code against the desired specifications and assessing its ability to produce the expected output [40].
12	**Completeness**	The generated code should cover all the necessary functionality and requirements specified by the given input. Evaluating completeness involves analyzing whether the generated code addresses all the required features and functionality, leaving no critical gaps or missing components. [40], [21].
13	**Maintainability**	Evaluating maintainability focuses on the ease of understanding, modifying, and extending the generated code. Factors such as code readability, modularity, and adherence to coding standards are considered. Assessing maintainability involves reviewing the code structure, documentation, and the availability of appropriate abstractions and encapsulation. [40], [63], [39], [21].

Figure 2 shows the frequency of most important evaluation criteria used in the selected research. For example, correctness, accuracy, maintainability, robustness, and consistency are highly used, while portability used with low frequency, and the others are medium frequently used.

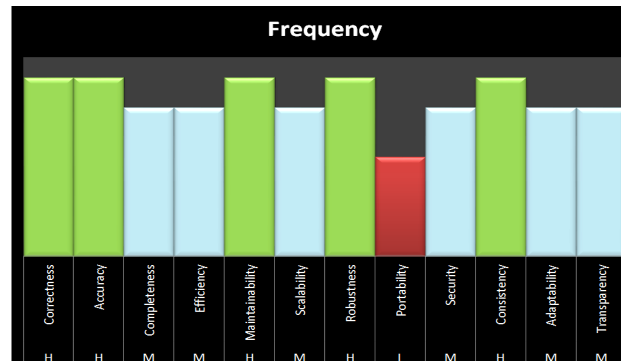


Figure. 2. Frequency of used criteria

5.3. Strengths, Weaknesses, and Performance

Depending on the evaluation criteria extracted from related researches, we were able to extract points of strength, weakness, and performance as shown in the Table 6.

Table 6. Comparative analysis of AIT for ACG

AIT	Strengths	Weaknesses	Performance
TB	• Simplicity and ease of use. • Well-suited for repetitive patterns	• Limited flexibility and adaptability. • Limited handling of complex logic.	• Fast code generation for specific tasks. • Limited scalability for diverse applications.
RB	• Explicit representation of domain knowledge. • Clear traceability of code generation decisions.	• Requires extensive rule definition. • Complexity increases with rule set size.	• Efficient for applications with clear rule structures. • Suitable for well-defined, rule-bound domains.
DL	• Ability to learn complex patterns from data. • Adaptability to diverse and evolving requirements.	• Dependence on quality and quantity of training data. • Lack of interpretability in decision-making.	• Good performance for tasks with sufficient training data. • Requires continuous training for changing contexts.
DL	• Capability to capture intricate syntax and semantics. • End-to-end learning for feature extraction.	• Demands large amounts of high-quality training data. • Computational complexity in training deep models.	• High accuracy in generating contextually rich code. • Superior performance for tasks with complex patterns.
EAs	• Exploration of diverse solution spaces. • Adaptive to changing problem landscapes.	• High computational overhead for complex problems. • Convergence to optimal solutions not always guaranteed.	• Effective for optimization-based code generation tasks. • Suitable for heuristic-driven, complex optimization tasks.
NLP	• Can be used to generate code from natural language descriptions.	• This can be a more natural way to generate code for non-technical users.	• Can be effective for generating code for simple tasks, but it is not as effective for generating code for complex tasks.

Please note that the performance of these approaches can vary based on the specific context and problem domain. Additionally, advancements in technology and methodologies could impact the strengths, weaknesses, and performance of these approaches in the future.

6. Future Directions and Research Opportunities

In this section, we hope to highlight potential pathways and unexplored avenues that merit further investigation and innovation.

By identifying these opportunities, we hope to chart a course for future research that will address existing challenges, leverage new technologies, and ultimately push the boundaries of what is possible in automated code generation. This forward-thinking exploration aims to shape the future rather than simply predict it.

6.1. Directions and Areas for Improvement in AIT for ACG

Based on the reviewed references, we found important potential research directions and areas for improvement in AIT for ACG such as: Integration of NLP and Code Generation, ML for Big Code and Naturalness, and others. The most important directions are provided in Table 7.

Table 7. Directions and areas for improvement in AIT for ACG

No	Directions	Improvement in AIT for ACG
1	Integration of NLP and Code Generation	Investigate methods to improve the understanding and translation of natural language descriptions into executable code [27], [29]. This could involve exploring techniques such as pre-training models with external knowledge [29], incorporating lexical and grammatical processing, and leveraging DL models for better source code modeling [7], [30], [31].
2	ML for Big Code and Naturalness	Explore ML approaches that leverage big code repositories to improve ACG [25]. This could involve using techniques like neural networks, DL, or transfer learning to capture patterns and generate code that aligns with the naturalness of human-written code
3	Code Generation from GUI Images	Investigate AIT to automatically generate code from graphical user interface (GUI) mock-up images [9], [54]. This area could involve developing computer vision models combined with DL techniques to recognize UI elements and generate corresponding code.
4	Code Generation using EAs	Explore the application of evolutionary algorithms in automating code generation [37], [48], [49]. This research direction could involve investigating how evolutionary algorithms can be used to generate optimized code and solve complex programming problems.
5	Improving AI Model Training and Evaluation	Improving AI models training and evaluation can help in addressing some challenges such as selection of dataset, generalization, and interpretability [2], [6].
6	TB Code Generation	This research direction could involve developing advanced template systems that can generate code based on predefined templates and adapt to different contexts or requirements.
7	ACG for Specific Domains	Exploring AITs for ACG in specific domains (industrial, self-adaptive, mobile) can improve these domains and algorithms used for generating code [22], [55].
8	Code Generation Tools and IDE Integration	This direction enhances developer productivity and support real-time code suggestions or completions [26], [43].
9	Software Maintenance and Refactoring Support	Investigate AIT for ACG that can assist in software maintenance tasks, such as refactoring or bug fixing [18].
10	DL -based Code Generation	DL-based architectures can capture complex patterns and dependencies in code and generate high-quality code snippets.
11	Performance Optimization	Performance optimization can enhance efficiency of the generated code, and performance metrics such as execution time, memory usage, or energy consumption.
12	Human-in-the-Loop Approaches	Explore approaches that incorporate human feedback and interactions into the code generation process. This can involve techniques such as interactive code completion, code refactoring suggestions, or collaborative code generation environments that combine human expertise with AI capabilities
13	Hybrid Approaches	Integrate AIT with traditional RB or TB methods to harness the advantages of both approaches.
14	Domain-Specific Languages (DSLs):	Design and implement DSLs that are optimized for code generation tasks in specific application domains. DSLs can provide high-level abstractions and specific syntax tailored to the domain, enabling more efficient and accurate code generation [25], [2]
15	Transfer Learning and Pre-training	Transfer learning can be used to improve code generation by pre-training models on large code repositories or related tasks.
16	Multi-modal Code Generation	Explore techniques that combine multiple modalities, such as code snippets, natural language descriptions, and diagrams, to generate code. This approach can improve the accuracy of generated code
17	Incremental Code Generation (ICG)	Create ICG algorithms, allowing developers to enhance generated code.

The ongoing research, in AITs aimed at ACG has the potential to greatly transform the field of software development. Although there are some obstacles to overcome the continuous efforts being made in this area are yielding progress. By prioritizing the suggested research directions and implementing improvements we can further enhance AIT for ACG making it more robust, effective, and user friendly.

7. Conclusion

Finally, in this work, various AITs for ACG in software development have been discussed and compared; it highlighted the limitations of traditional and AI approaches, and explored the introduction of AI techniques in code generation. Through our analysis, we have identified the strengths and weaknesses of different AITs, including RB systems, ML, and DL.

These techniques have shown significant advancements in generating code, improving efficiency, and enhancing code quality. We have also discussed their relevance and applicability to different code generation tasks. Furthermore, we have presented recent developments and innovations in AI techniques for code generation, showcasing case studies and real-world applications that demonstrate their effectiveness. The evaluation metrics commonly used to assess the performance of AI-based code generation systems were discussed as well. While AI techniques offer great potential in automated code generation, they come with their own challenges and limitations. Data requirements and availability for training AI models, scalability, efficiency, and the interpretability of AI-generated code are among the key challenges that need to be addressed.

These considerations will shape the adoption and integration of AI techniques in real-world software development. Based on our comparative analysis, we have provided insights into the strengths and weaknesses of each AI technique, enabling researchers and practitioners to make informed decisions in selecting and combining methodologies based on specific requirements and constraints. The implications of AI-based code generation in software development are significant. By automating repetitive and time-consuming code generation tasks, developers can focus more on higher-level design and critical problem-solving. This can lead to increased productivity, accelerated software development cycles, and improved software quality.

References:

[1]. Meziane, F., & Vadera, S. (2010). Artificial intelligence in software engineering: current developments and future prospects. *Artificial intelligence applications for improved software engineering development: New prospects*, 278-299.

[2]. Dehaerne, E., Dey, B., Halder, S., De Gendt, S., & Meert, W. (2022). Code generation using machine learning: A systematic review. *Ieee Access*, *10*, 82434–82455. Doi: 10.1109/ACCESS.2022.3196347.

[3]. Shahzad, B., Abdullatif, A. M., Ikram, N., & Mashkoor, A. (2017). Build software or buy: A study on developing large scale software. *IEEE Access*, *5*, 24262-24274. doi: 10.1109/ACCESS.2017.2762729.

[4]. P. M. Khan and M. M. Sufyan Beg. (2012). Measuring Cost of Quality(CoQ)- on SDLC projects is indispensible for effective Software Quality Assurance. *International Journal of Soft Computing and Software Engineering, 2*(9), 1–15. Doi: 10.7321/JSCSE.V2.N9.1.

[5]. Chemnitz, L., Reichenbach, D., Aldebes, H., Naveed, M., Narasimhan, K., & Mezini, M. (2023). Towards Code Generation from BDD Test Case Specifications: A Vision. *Proc. - 2023 IEEE/ACM 2nd Int. Conf. AI Eng. - Softw. Eng*. 139–144. Doi: 10.1109/CAIN58948.2023.00031.

[6]. Yang, Z., Chen, S., Gao, C., Li, Z., Li, G., & Lv, R. (2023). Deep Learning Based Code Generation Methods: A Literature Review. *arXiv preprint arXiv:2303.01056*.

[7]. Le, T. H., Chen, H., & Babar, M. A. (2020). Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, *53*(3), 1-38. Doi: 10.1145/3383458.

[8]. Zhang, X., Jiang, Y., & Wang, Z. (2019). Analysis of automatic code generation tools based on machine learning. In *2019 IEEE International Conference on Computer Science and Educational Informatization (CSEI)*, 263-270. IEEE. Doi: 10.1109/CSEI47661.2019.8938902.

[9]. Aşıroğlu, B., Mete, B. R., Yıldız, E., Nalçakan, Y., Sezen, A., Dağtekin, M., & Ensari, T. (2019). Automatic HTML code generation from mock-up images using machine learning techniques. In *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT)*, 1-4. IEEE. Doi: 10.1109/EBBT.2019.8741736.

[10]. Yang, C., Liu, Y., & Yin, C. (2021). Recent Advances in Intelligent Source Code Generation: A Survey on Natural Language Based Studies. *Entropy*, *23*(9), 1174. Doi: 10.3390/E23091174.

[11]. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., & Sarro, F. (2021). A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610*.

[12]. Zhang, C., Niu, X., & Yu, B. (2018). A method of automatic code generation based on AADL model. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, 180-184. Doi: 10.1145/3297156.3297172.

[13]. Koziolek, H., Burger, A., Platenius-Mohr, M., Rückert, J., Abukwaik, H., Jetley, R., & P, A. P. (2020). Rule-based code generation in industrial automation: four large-scale case studies applying the cayenne method. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 152-161. Doi: 10.1145/3377813.3381354.

[14]. Soliman, A. S., Hadhoud, M. M., & Shaheen, S. I. (2022). MarianCG: A code generation transformer model inspired by machine translation. *Journal of Engineering and Applied Science*, *69*(1), 1-23.

[15]. Bilgin, Z. (2021). Code2image: Intelligent code analysis by computer vision techniques and application to vulnerability prediction. *arXiv preprint arXiv:2105.03131*.

[16]. Arogundade, O. T., Onilede, O., Misra, S., Abayomi-Alli, O., Odusami, M., & Oluranti, J. (2021). From modeling to code generation: an enhanced and integrated approach. In *Innovations in Information and Communication Technologies (IICT-2020) Proceedings of International Conference on ICRIHE-2020, Delhi, India: IICT-2020*, 421-427. Springer International Publishing.

[17]. Chen, H. (2020). Design and implementation of automatic code generation method based on model driven. In *Journal of Physics: Conference Series*, *1634*(1), 012019. IOP Publishing. Doi: 10.1088/1742-6596/1634/1/012019.

[18]. Nagulapati, V., Rapelli, S. R., & Fiaidhi, J. (2020). Automating Software Development using Artificial Intelligence. *TechRxiv*. Doi: 10.36227/TECHRXIV.12089139.V1.

[19]. Dahal, S., Maharana, A., & Bansal, M. (2021). Analysis of tree-structured architectures for code generation. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 4382-4391. Doi: 10.18653/V1/2021.FINDINGS-ACL.384.

[20]. Yu, P., Shu, H., Xiong, X., & Kang, F. (2021, December). A random code generation method based on syntax tree layering model. In *International Conference on Electronic Information Engineering and Computer Technology (EIECT 2021)*,*12087*, 465-476. SPIE. Doi: 10.1117/12.2624688.

[21]. Kumari, V. I. P. A. N., & Kulkarni, S. A. N. D. E. E. P. (2018). Use of artificial intelligence in software development life cycle requirements and its model. *Int. Res. J. Eng. Technol.(IRJET)*, *5*(8), 398-403.

[22]. Lee, J., Park, J., Yoo, G., & Lee, E. (2010). Goal-based automated code generation in self-adaptive system. *Journal of Computer Science and Technology*, *25*(6), 1118-1129. Doi: 10.1007/S11390-010-9393-2/METRICS.

[23]. Imam, A. T., Rousan, T., & Aljawarneh, S. (2014). An expert code generator using rule-based and frames knowledge representation techniques. In *2014 5th International Conference on Information and Communication Systems (ICICS)*, 1-6. IEEE. Doi: 10.1109/IACS.2014.6841951.

[24]. Pont, M. J. (2014). Prolog as a Language for Rule-Based Code Generation. *Theory Pract. Log. Program, 5*.

[25]. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 1-37. Doi: 10.1145/3212695.

[26]. Desai, A., & Deo, A. (2022). *Introducing Amazon CodeWhisperer, the ML-powered coding companion* . AWS Machine Learning Blog. Retrieved from: https://aws.amazon.com/blogs/machine-learning/introducing-amazon-codewhisperer-the-ml-powered-coding-companion/ [accessed: 14 September 2023.].

[27]. Zhu, J., & Shen, M. (2020). Research on Deep learning Based Code generation from natural language Description. In *2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, 188-193. IEEE. Doi: 10.1109/ICCCBDA49378.2020.9095560.

[28]. Beau, N., & Crabbé, B. (2022). The impact of lexical and grammatical processing on generating code from natural language. In *Findings of the Association for Computational Linguistics: ACL 2022*, 2204–2214, Dublin, Ireland. Association for Computational Linguistics.

[29]. Xu, F. F., Jiang, Z., Yin, P., Vasilescu, B., & Neubig, G. (2020). Incorporating External Knowledge through Pre-training for Natural Language to Code Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 6045–6052, Association for Computational Linguistics. Doi: 10.18653/V1/2020.ACL-MAIN.538.

[30]. Hussain, Y., Huang, Z., Zhou, Y., & Wang, S. (2020). CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. *Information and Software Technology*, *125*, 106309. Doi: 10.1016/J.INFSOF.2020.106309.

[31]. Hussain, Y., Huang, Z., Zhou, Y., & Wang, S. (2020). Deep transfer learning for source code modeling. *International Journal of Software Engineering and Knowledge Engineering*, *30*(05), 649-668. Doi: 10.1142/S0218194020500230.

[32]. Priya, R., Wang, X., Hu, Y., & Sun, Y. (2017). A deep dive into automatic code generation using character based recurrent neural networks. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, 369-374. IEEE. Doi: 10.1109/CSCI.2017.61.

[33]. Saravanan, S., & Sudha, K. (2022). GPT-3 powered system for content generation and transformation. In *2022 Fifth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, 514-519. IEEE. Doi: 10.1109/CCICT56684.2022.00096.

[34]. Liu, C., Bao, X., Zhang, H., Zhang, N., Hu, H., Zhang, X., & Yan, M. (2023). Improving ChatGPT Prompt for Code Generation. *arXiv preprint arXiv:2305.08360*.

[35]. Mitzalis, F., Caglayan, O., Madhyastha, P., & Specia, L. (2021). BERTGen: Multi-task Generation through BERT, In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing,1,Long Papers*, 6440–6455.

[36]. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., ... & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI open*, *1*, 57-81. Doi: 10.1016/J.AIOPEN.2021.01.001.

[37]. Mironovich, V., Buzdalov, M., & Vyatkin, V. (2017). Automatic generation of function block applications using evolutionary algorithms: Initial explorations. In *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 700-705. IEEE. Doi: 10.1109/INDIN.2017.8104858.

[38]. Cortes, O. A., Eveline de Jesus, V. S., da Silva, J. A., & Rau-Chaplin, A. (2015). An Automatic Code Generator for Parallel Evolutionary Algorithms: Achieving Speedup and Reducing the Programming Efforts. *ADVCOMP 2015*, 48.

[39]. Insaurralde, C. C. (2013). Software programmed by artificial agents toward an autonomous development process for code generation. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, 3294-3299. IEEE. Doi: 10.1109/SMC.2013.561.

[40]. Simonsen, K. I. F. (2014). An evaluation of automated code generation with the PetriCode approach. In *CEUR Workshop Proceedings*, 289–306.

[41]. Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., & Prather, J. (2022). The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, 10-19. Doi: 10.1145/3511861.3511863.

[42]. Rishi. (2023). *ChatGPT – An Insight To Fun Facts For All Data Scientists*. Education. Retrieved from: https://vocal.media/education/chat-gpt-an-insight-to-fun-facts-for-all-data-scientists [accessed: 13 September 2023].

[43]. Cruz-Benito, J., Vishwakarma, S., Martin-Fernandez, F., & Faro, I. (2021). Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches. *AI*, *2*(1), 1-16. Doi: 10.3390/AI2010001.

[44]. Yetistiren, B., Ozsoy, I., & Tuzun, E. (2022). Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 62-71. Doi: 10.1145/3558489.3559072.

[45]. Shim, S., Patil, P., Yadav, R. R., Shinde, A., & Devale, V. (2020). DeeperCoder: Code Generation Using Machine Learning. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 0194-0199. IEEE. Doi: 10.1109/CCWC47524.2020.9031149.

[46]. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with alphacode. *Science*, *378*(6624), 1092-1097. Doi: 10.1126/SCIENCE.ABQ1158.

[47]. I. S. Bajwa, M. I. Siddique, and M. A. Choudhary. (2006). Rule based production systems for automatic code generation in Java, 2006 1st Int. Conf. Digit. Inf. Manag. ICDIM, 300–305, Doi: 10.1109/ICDIM.2007.369214.

[48]. Sobania, D., Schweim, D., & Rothlauf, F. (2022). Program synthesis with evolutionary algorithms: Status quo: hot off the press track (GECCO 2022). In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 39-40. Doi: 10.1145/3520304.3534074.

[49]. Sobania, D., Schweim, D., & Rothlauf, F. (2022). A comprehensive survey on program synthesis with evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, *27*(1), 82-97. Doi: 10.1109/TEVC.2022.3162324.

[50]. Hu, K., Duan, Z., Wang, J., Gao, L., & Shang, L. (2019). Template-based AADL automatic code generation. *Frontiers of Computer Science*, *13*, 698-714.

[51]. Danilchenko, Y., & Fox, R. (2012). Automated code generation using case-based reasoning, routine design and template-based programming. In *Midwest Artificial Intelligence and Cognitive Science Conference*, 119-125.

[52]. Pinto-Santos, F., Alizadeh-Sani, Z., Alonso-Moro, D., González-Briones, A., Chamoso, P., & Corchado, J. M. (2021). A template-based approach to code generation within an agent paradigm. In *Highlights in Practical Applications of Agents, Multi-Agent Systems, and Social Good. The PAAMS Collection: International Workshops of PAAMS 2021, Salamanca, Spain, October 6–9, 2021, Proceedings 19*, 296-307. Springer International Publishing.

[53]. Google. (n.d.). *Google Cloud AutoML - Train models without ML expertise*. Cloud google. https://cloud.google.com/automl/ [accessed: 18 September 2023.].

[54]. de Souza Baulé, D., von Wangenheim, C. G., von Wangenheim, A., & Hauck, J. C. (2020). Recent Progress in Automated Code Generation from GUI Images Using Machine Learning Techniques. *J. Univers. Comput. Sci.*, *26*(9), 1095-1127. Doi: 10.3897/JUCS.2020.058.

[55]. Baulé, D., von Wangenheim, C. G., von Wangenheim, A., Hauck, J. C., & Júnior, E. C. V. (2021). Automatic code generation from sketches of mobile applications in end-user development using Deep Learning. *arXiv preprint arXiv:2103.05704*.

[56]. Tiwang, R., Oladunni, T., & Xu, W. (2019). A deep learning model for source code generation. In *2019 SoutheastCon*, 1-7. IEEE. Doi:10.1109/SOUTHEASTCON42311.2019.9020360

[57]. Aşıroğlu, B., Mete, B. R., Yıldız, E., Nalçakan, Y., Sezen, A., Dağtekin, M., & Ensari, T. (2019). Automatic HTML code generation from mock-up images using machine learning techniques. In *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT)*, 1-4. IEEE. Doi: 10.1109/EBBT.2019.8741736.

[58]. Lee, C., Gottschlich, J., & Roth, D. (2021). Toward code generation: A survey and lessons from semantic parsing. *arXiv preprint arXiv:2105.03317*.

[59]. Zhu, Q. et al. (2021). Code Generation Based on Deep Learning: a Brief Review. *ESEC/FSE 2021 - Proc. 29th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 21*, 341–353.

[60]. Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023). Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* , 500-506. Doi: 10.1145/3545945.3569759.

[61]. cybertechworld.co.in. (2023). *Artificial Intelligence and Machine Learning in Cybersecurity.* Cybertechworld. Retrieved from: https://cybertechworld.co.in/artificial-intelligence-and-machine-learning/ [accessed: 20 September 2023].

[62]. Korzeniowski, Ł., & Goczyła, K. (2019). Artificial intelligence for software development: the present and the challenges for the future. *Biuletyn Wojskowej Akademii Technicznej*, *68*(1). Doi: 10.5604/01.3001.0013.1464.

[63]. Wangoo, D. P. (2018). Artificial intelligence techniques in software engineering for automated software reuse and design. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)*, 1-4. IEEE. Doi: 10.1109/CCAA.2018.8777584.