Exploring AI Innovations in Automated Software Source Code Generation: Progress, Hurdles, and Future Paths

Ayman Odeh*, Nada Odeh
Software Engineering and Computer Science Department, College of Engineering, Al Ain University, Al Jimi, Al Ain, UAE
E-mail: ayman.odeh@aau.ac.ae
*Corresponding author

Keywords: artificial intelligence, automated source code generation, deep learning, evolutionary algorithms, machine learning, natural language processing

Received: October 16, 2023

In today's dynamic world of software development, the demand for efficient and rapid creation of high-quality code has never been more pronounced. Automated software source code generation (ASSCG) emerges as a compelling solution to meet this demand, offering significant advantages in terms of speed, accuracy, and scalability. This paper aims to explore the critical role of automated software source code generation and its profound significance in modern software development practices. By navigating through the intersection of ASSCG, AI innovations, and the challenges therein, this paper endeavors to provide a comprehensive understanding of this transformative field and pave the way for informed decision-making and advancements in software development practices. This paper delves into the critical role of ASSCG and its transformative impact on modern software development. In this work, we endeavor to delve into the multifaceted landscape of automated code generation, assessing its significance, the transformative potential of AI innovations, and the challenges and objectives inherent in this evolving domain.

Povzetek: Prispevek raziskuje vloge in izzive avtomatizirane generacije programske kode z uporabo umetne inteligence. Poudarek je na inovacijah z uporabo globokega učenja in evolucijskih algoritmov, ki izboljšujejo hitrost in natančnost kodiranja. Študija identificira ključne ovire in prihodnje smeri razvoja.

1 Introduction

In the realm of modern software development, the quest for efficiency and agility has led to a burgeoning interest in automated software source code generation. This introduction sets the stage for a comprehensive exploration of the topic within the context of AI innovations. ASSCG stands as a cornerstone of modern software development methodologies, offering the promise of accelerated development cycles, enhanced productivity, and improved code quality. These technologies speed up creativity and product delivery by automating time-consuming, repetitive chores like boilerplate generation and code scaffolding. In this case developers can now focus on more complex design and problem-solving.

Central to the advancements in automated code generation is the integration of AI technologies, which have ushered in a new era of possibilities in software engineering. From machine learning algorithms capable of predicting code patterns to natural language processing techniques facilitating code synthesis from human-readable specifications, AI innovations are reshaping the way software is abstracted, developed, and implemented. This overview sets the stage for a deeper exploration of the transformative potential of AI in revolutionizing the software engineering landscape [1].

However, amidst the optimism surrounding AI-driven automated code generation, several challenges and complexities arise, necessitating a nuanced understanding of the underlying issues. The statement of the problem in this paper revolves around identifying these hurdles, including issues related to algorithmic biases, code quality assurance, and the ethical implications of AI-generated code. With these challenges in mind, the objectives of this paper are twofold: to critically assess the current state of AI innovations in ASSCG and to chart a path forward that addresses these challenges while maximizing the potential benefits of AI-driven development tools.

1.1 ASSCG significance in software development

Automated code generation acts as a supercharger for software development, offering a multitude of benefits. From rapid development through automatic code creation to improved quality via consistent style and reduced errors, it frees developers for complex tasks. Furthermore, it simplifies maintenance, fosters collaboration through clear interfaces, and reduces overall costs. By enabling faster prototyping and knowledge transfer through standardized code,

automated code generation empowers developers to adapt to changing project requirements and validate ideas quicker.

1.2 The problem statement

In this context revolves around understanding the nuances of ASSCG, including its limitations, potential biases, and implications for software quality and maintainability. Furthermore, this paper aims to delineate clear objectives aimed at addressing these challenges and harnessing the full potential of automated code generation tools in practical software engineering scenarios.

1.3 Objectives

By embarking on this exploration, the paper aims to provide valuable insights into the evolving landscape of automated code generation, offering guidance to researchers, practitioners, and stakeholders navigating the intersection of AI and software engineering. Through a balanced examination of progress, hurdles, and future paths, this paper seeks to contribute to the advancement of ASSCG and its integration into mainstream software development practices.

1.4 The paper structure

Section 4 provides Overview of AI Innovations in ASSCG, in section 4, we provide identification and analysis of challenges faced in AI-driven ASSCG, section 5 presents several case studies, in Section 6, we provide Future Paths and Research Opportunities, and finally, the conclusion was provided in section 7.

2 Literature review

This section will give a summary of the conventional approaches to software source code generation (SSCG), a review of the literature on AI advances applied to ASSCG, and a quick analysis of the effects of AI on ASSCG.

2.1 Traditional methods of (SSCG).

Traditional methods of software source code generation have been foundational to the evolution of software engineering practices. These methods typically involve manual coding by developers using text editors or integrated development environments (IDEs). For example, **Hand Coding** requires deep knowledge of both the chosen programming language (C, Java, Python, etc.) and the specific problem they were solving. This approach, while foundational, can be time-consuming and error-prone. **Copy-Pasting:** In situations where similar functionality is required in multiple parts of a project, developers may resort to copy-pasting existing code snippets and making necessary modifications. While this approach can save time, it often leads to code redundancy and maintenance issues. **Code Templates** [2]**:** Another traditional method involves the use of code templates or boilerplate code provided by IDEs or libraries. Developers can use these templates as a starting point and customize them according to their requirements. However, this method still requires manual intervention for adaptation and customization. **Code Generators:** Some traditional software development environments include code generators that automate the generation of repetitive or boilerplate code. These generators may be built into the IDE or provided as separate tools. They typically operate based on predefined templates or rules provided by the developer [3][4]. **Scripting:** In certain cases, developers may employ scripting languages or scripting tools to automate specific tasks within the software development process. These scripts can automate repetitive tasks, such as file manipulation, data processing, or code generation based on certain criteria. **Manual Refactoring** [5]**:** When existing code needs to be optimized or modified to improve readability, performance, or maintainability, developers may engage in manual refactoring. This involves restructuring existing code without changing its external behavior, often to make it more efficient or easier to understand.

While these traditional methods have been instrumental in software development for decades, they are often labor-intensive, error-prone, and time-consuming. With the emergence of ASSCG techniques, there has been a shift towards leveraging AI and machine learning to streamline and enhance the code generation process, leading to increased efficiency and productivity in software development.

2.2 Related works

A range of AI techniques have been applied to ASSCG, each with its own strengths and weaknesses. Yen [6] proposed the use of AI planning techniques to synthesize glue code and automate testing, while Cruz-Benito [7] compared the performance of different deep learning architectures in generating code. Danilchenko [8] introduced a system that combined Case-Based Reasoning, Routine Design, and Template-Based Programming [8] for ASSCG, and Dehaerne[5] provided a systematic review of studies using machine learning for code generation, highlighting the use of recurrent neural networks, transformers, and convolutional neural networks. These studies collectively demonstrate the potential of AI in code generation, with each approach offering unique contributions and considerations. While rule-based methods like Koziolek et al.'s [9] in industrial automation shine in reliability and predictability, their rigidity can pose challenges for handling complex, non-standardized tasks. Meanwhile, natural language processing (NLP) integration, as highlighted by Zhu and Shen's [10] work on natural language to code generation, unlocks exciting possibilities but necessitates models attuned to the intricate context of software development language. Finally, evolutionary algorithms, exemplified by Mironovich et al.'s [11]work on function block applications, offer a unique approach to optimizing code but face scalability concerns and limitations in real-time scenarios. Recent research has focused on the use of deep

learning in programming language models, particularly for tasks such as auto-completion and code generation. Cruz-Benito (2020) [8] compares different neural network architectures, including AWD-LSTMs, AWD-QRNNs, and Transformer, for building language models using a Python dataset. The study highlights the strengths and weaknesses of each approach and identifies gaps in evaluating and applying these models in a real programming context. Le[12] offers a thorough analysis of the deep learning techniques currently in use for generating and modeling source code, categorizing program learning tasks and discussing the challenges and recommendations for practitioners and researchers. These studies collectively underscore the potential of deep learning in software engineering and the need for further research in this area. Another area of focus is the automatic generation of source code comments, with a survey of algorithms and techniques [13]. Finally, a component-based approach has been proposed for the systematic generation of correct, compatible, and efficient database structures and manipulation function modules [14].

In Table1, we provide the key findings from the surveyed literature, focusing on evaluation metrics such as accuracy, efficiency, scalability, generalization. Where these metrics values are: High: (H), Limited (Lm), Low (Lo), Moderate (M), and Conceptual: (Co).

Table 1: ASSCG Evaluation Metrics Summary Table

No	Model / Technique	Accuracy	Efficiency	Scalability	Generalizability	Correctness
1	Machine Learning Survey	-	-	-	-	- Analyzed strengths & weaknesses of various ML approaches. - Highlighted ML techniques for code analysis, including generation.
2	Deep Learning Survey	-	-	-	-	Explored deep learning models & challenges in code modeling/generation.
3	Rule-based (CAYENNE)	H	-	-	Lm	Focused on industrial automation code generation.
4	Code2Image (Computer Vision)	-	-	-	-	Promising for vulnerability prediction; limited code generation capabilities.
5	Random Code Generation (Syntax Tree)	Lo	H	-	-	Fast generation, but often syntactically incorrect or nonsensical.
6	AI in Software Development	-	-	-	-	Conceptual overview; no empirical evaluation.
7	Goal-based Code Generation	-	-	-	-	Focused on self-adaptive systems; limited evaluation details.
8	Expert Rule-based & Frames	-	-	-	-	Knowledge-intensive; unclear generalizability or correctness.
9	Prolog for Rule-based Generation	-	-	-	-	Primarily focused on logic programming; limited empirical data.
10	Machine Learning for Big Code & Naturalness	-	-	-	-	Surveyed existing ML approaches for code generation.
11	Amazon CodeWhisperer	-	-	-	-	Proprietary ML-powered coding assistant; limited public evaluation.
12	Deep Learning from Natural Language Descriptions	M	-	-	-	Promising results, but requires large datasets and careful training.
13	Lexical & Grammatical Processing for Code from NL	M	-	-	-	Improved accuracy by focusing on language aspects, but still challenging.
14	Pre-training with External Knowledge for NL2Code	H	-	-	Lm	Demonstrated improved accuracy with external knowledge integration.
15	Context-aware Deep Learning with GRU (CodeGRU)	H	-	-	Lm	Achieved high accuracy on specific tasks, but generalizability unclear.
16	Deep Transfer Learning for Code Modeling	H	M	-	Lm	Effective for code similarity tasks, but generation capabilities not explored.
17	Character-based Recurrent Neural Networks	M	M	-	Lm	Can generate code snippets, but often lacks context and coherence.
18	GPT-3 for Content Generation and Transformation	-	-	-	-	Demonstrated potential for code generation, but limited technical details.
19	Improved ChatGPT Prompts for Code Generation	M	-	-	Lm	Showed promising results with fine-tuned prompts, but further research needed.
20	BERTGen: Multi-task Generation through BERT	H	-	-	Lm	Achieved state-of-the-art accuracy on specific benchmarks, but generalizability unknown.
21	Evolutionary Algorithms for Function Block Applications	M	M	-	Lm	Explored automatic generation of code blocks, but scalability and correctness not thoroughly evaluated.
22	Automatic Code Generator for Parallel Evolutionary Algorithms	M	M	Lm	Lm	Demonstrated speedup and reduced programming effort, but code quality and generalizability unclear.
23	Artificial Agents (AG) software generated [15]	Co	-	-	-	suggested an independent development approach but lacked specific execution and assessment.
24	AI Planning Techniques for Automated Code Synthesis & Testing	Lo	-	-	Lm	Pioneered the use of AI planning for code generation, but accuracy and efficiency were low.
25	Deep Learning for auto-g & Auto-completion	M	M	-	Lm	Compared and discussed deep learning approaches, but lacked in-depth evaluation of specific models.
26	Case-based Reasoning, Routine	M	M	Lm	Lm	Combined different techniques for code generation, but scalability

	Design, & Programming[16]	Template-based and correctness not extensively studied.

2.2 Current State-of-the-Art (SOTA) techniques

This paper explores the SOTA in AI-driven Automatic Software Source Code Generation (ASSCG), examining promising techniques like Deep Learning (LSTMs and Transformers for natural language to code translation), Rule-based Methods (CAYENNE for predictable code in industrial automation), Evolutionary Algorithms (optimizing code for specific goals), and AI Planning Techniques (frameworks like STRIPS for autonomous code generation). However, limitations like limited domain specificity, accuracy, and bias, alongside interpretability challenges, remain. By analyzing a list of these techniques, and providing future research directions, this paper aims to address these limitations and promote responsible development of ASSCG, ultimately unlocking its full potential for software development.

3 AI Innovations for ASSCG

ASSCG has experienced remarkable advancements fueled by various AI innovations. Let's explore each of these innovations in detail:

Rule-based Systems (RBS): These systems create code from high-level specifications using predetermined rules and patterns. Frequently utilized in specialized fields and industrial automation, RBS provide control and interpretability over the code that is created. [9] [17]. Machine Learning (ML) Approaches: ML tools, including transformers, RNNs, and statistical language models, are trained on large code repositories to identify patterns and correlations, which allows them to produce code based on learned information.[5] [18][19]. Natural Language Processing (NLP) Techniques: NLP enables code generation from natural language descriptions via the use of describing, neural language models, and StoSMs. Code creation skills are improved by NLP-based techniques by incorporating external knowledge sources. [10] [18] [20]. Deep Learning (DL) Models: DL models, including RNNs, transformers like GPT and BERT, and GNNs, excel in modeling and generating source code. Transformers are particularly useful due to their context-awareness, attending to relevant code contexts during generation [10][18][21][22][23][24]. Evolutionary Algorithms (EAs): EAs offer an alternative approach by iteratively mutating and evolving existing code to generate new solutions. They have been utilized to produce efficient, maintainable code across different programming languages, platforms, and applications [25] [26]. Autonomous Method (AM) [14] [27] [28]: Autonomous code generation is a fascinating area of research and development that uses artificial intelligence (AI) techniques to automatically write software code. It has the potential to revolutionize software development by speeding up development processes, reducing errors, and increasing productivity.

The explanation of how each innovation works and its application in generating software source code will be provided in Table 2, and some examples of applications of AI innovation in ASSCG will be listed in Table 3:

Table 2: Overview of ai innovations in automated software source code generation

Innovation	How it Works	Application
Rule-based Systems (RBS)	Relies on predefined rules and patterns to transform high-level requirements into executable code.	Commonly used in industrial automation and specific application domains with well-defined processes.
Machine Learning (ML)	Using large scale code repositories in training process to learn patterns and relationships, generating code based on the learned knowledge.	Versatile and can be applied to various code generation tasks, such as auto-completion and summarization.
Natural Language Processing (NLP)	Analyzes linguistic structure of natural language descriptions and translates them into executable code.	Facilitates code generation from natural language specifications, allowing for more intuitive expression of requirements.
Deep Learning (DL)	Learns complex relationships and patterns in code data, generating code snippets or complete functions based on the learned knowledge.	Effective for tasks like code summarization, auto-completion, and generation from natural language descriptions.
Evolutionary Algorithms (EAs)	Iteratively mutates and evolves existing code to generate new solutions, starting with a population of randomly generated code.	Useful for generating efficient, maintainable code across different programming languages and platforms.
Autonomous Method (AM)	Autonomous code generation (ACG) leverages artificial intelligence (AI) to automatically write software code, presenting a paradigm shift in software development. Its potential to boost efficiency, reduce errors, and enhance productivity makes it a hot topic in research and development.	Simple function generation: Tools like DeepCode and Github Copilot utilize large language models (LLMs) to generate basic functions based on natural language descriptions or code snippets. Code completion: Frameworks like Tabnine and Kite recommend relevant code based on context and developer intent, streamlining coding workflows. Automated bug fixes: Systems like DeepFix rely on deep learning to detect and propose fixes for common programming errors.

Table 3: Applications of AI innovations in ASSCG

Application	Innovation(s) Involved
Industrial automation	RBS
Auto-completion and code summarization	ML, DL, AM
Code generation from natural language descriptions	NLP, DL
Optimization of existing code	ML, EAs
Generating efficient and maintainable code	RBS, EAs
Specific application domains with well-defined processes	RBS

4 Challenges and limitations

4.1 Challenges

AI-driven ASSCG faces several challenges related to data quality, complexity, ambiguity, context awareness, ethical considerations, and performance. Addressing these challenges requires interdisciplinary efforts, including advancements in AI research, data management, software engineering practices, and regulatory frameworks. By addressing these challenges, developers can harness the full potential of AI-driven code generation to enhance productivity, code quality, and innovation in software development. Table 4 provides a concise overview of the challenges faced in AI-driven ASSCG, along with their identification and analysis. Addressing these challenges is crucial for maximizing the effectiveness and reliability of AI-driven code generation systems.

Table 4: Summary of identification and analysis of AI ASSCG challenges

Challenges	Identification	Analysis
Data Quality and Quantity [19] [5] [4]	AI models require large, high-quality training data.	Limited availability of diverse, well-annotated code datasets can hinder model performance and lead to biased or inaccurate code generation.
Complexity and Maintainability [29]	Generated code may be overly complex or difficult to maintain.	Complexity in generated code can impede collaboration, increase the risk of errors during maintenance, and hinder scalability of the codebase.
Ambiguity and Natural Language Understanding [30]	Extracting precise requirements from natural language can be challenging.	Natural language inputs may contain ambiguity, context-dependency, and linguistic nuances, leading to inaccuracies or misinterpretations in code generation.
Lack of Context Awareness	AI models may lack contextual understanding of code semantics and constraints.	Contextually inappropriate code generation can result in inefficiencies, inconsistencies, and compatibility issues in the generated codebase.
Ethical and Legal Considerations [31][3]	AI-driven code generation raises concerns regarding intellectual property and ethics.	Developers must navigate ethical and legal considerations surrounding ownership, licensing, and misuse of AI-generated code.
Performance and Scalability[23][32]	AI models may exhibit performance bottlenecks or scalability limitations.	Optimizing efficiency and scalability of AI-driven code generation systems is crucial for timely and cost-effective development.

4.2 Limitations

While AI-driven ASSCG offers significant benefits in terms of efficiency, productivity, and innovation, it also faces limitations related to scalability, interpretability, and handling of complex software requirements as shown in Table 5. Overcoming these l limitations requires interdisciplinary efforts, including advancements in AI research, software engineering practices, and collaboration between developers and AI systems. By addressing these limitations, developers can harness the full potential of AI-driven code generation to accelerate software development and enhance code quality and reliability.

Table 5: Limitations of ASSCG

Limitation	Description	Analysis
Scalability [33]	While AI models may perform well on small-scale projects or specific tasks, they may encounter challenges when applied to large-scale software development projects or complex codebases.	Scaling AI code generation for large projects is difficult. It requires a lot of computing power and storage to handle the complex interactions between different code parts as the project grows. This complexity makes it hard for AI models to maintain good performance and generate accurate code.
Interpretability[34]	AI-generated code is like a magic trick - it works, but you don't know how. This lack of understanding makes it hard for developers to trust the code and fix issues if they arise	AI-generated code is like a black box - developers can't understand how it works. This makes it hard to trust the code, fix problems, and collaborate on projects using the code.
Handling of Complex Software Requirements [35]	AI for code generation isn't great with complex projects. It struggles to understand the specific rules (business	Current AI models for automated code generation exhibit limitations when applied to complex software development. The intricate decision-making processes

	logic) and specialized knowledge needed for those projects. Additionally, it has trouble with requirements that focus on things other than the code itself (non-functional requirements).	and interwoven functionalities inherent in such projects pose challenges for AI to capture effectively. This can result in incomplete or suboptimal code generation. Furthermore, the prevalence of edge cases and unexpected scenarios in complex software creates additional hurdles for AI to navigate.

5 Case studies and impact on software practices

To illustrate the potential of ASSCG, this section presents several case studies gleaned from the reviewed literature. We will also explore the impact of ASSCG on software development practices.

5.1 Case Studies

1) Industrial Automation with CAYENNE [9] (Rule-based):
   Project provide by Koziolek et al. (2020) used CAYENNE [9], a rule-based system, to generate code for industrial automation tasks in four large-scale case studies.
   Results: CAYENNE achieved high accuracy and reduced development time by up to 50%. However, it required significant upfront investment in rule creation and was limited to specific domains.
   Key takeaway: Rule-based approaches can be effective for generating code in specific domains when accuracy and maintainability are crucial.
2) Code2Image [36] for Vulnerability Prediction (Computer Vision):
   Project: Bilgin (2021) explored using Code2Image, a computer vision technique, to analyze code and predict vulnerabilities.
   Results: Code2Image showed promise in identifying potential vulnerabilities, but its code generation capabilities were limited.
   Key takeaway: Combining different AI techniques like computer vision and ML can offer new insights into code analysis and vulnerability detection.
3) Amazon CodeWhisperer [3] (Proprietary ML-powered Coding Assistant):
   Project: Desai and Atul (2022)[3] introduced Amazon CodeWhisperer, a proprietary ML-powered coding assistant that suggests code completions and snippets.
   Results: While details are limited, CodeWhisperer reportedly improves developer productivity and reduces coding errors.
   Key takeaway: Proprietary AI-powered tools are emerging to aid developers, but their inner workings and long-term impact remain to be seen.
4) GPT-3 [23] for Content Generation and Transformation (Fine-tuning Prompts):
   Project: Liu et al. (2023) [22] investigated fine-tuning prompts for GPT-3, a powerful language model, to improve its code generation capabilities.
   Results: Fine-tuned prompts yielded moderate improvements in code accuracy and fluency compared to generic prompts.
   Key takeaway: Fine-tuning large language models like GPT-3 shows promise for code generation but requires careful prompt design and further research.
5) GitHub Copilot: GitHub Copilot is a tool that uses artificial intelligence to help developers write code. It can suggest code completions, generate entire functions, and even write entire programs [37].
6) Google Cloud AutoML Code: It can be used to generate code for a variety of PLs and platforms[38] using ML.

These are just a few examples. The field of code generation is rapidly evolving, with new techniques and applications emerging all the time. As AI research progresses, we can expect to see even more sophisticated and efficient code generation tools in the future.

5.2 Impact on software practices

The impact of AI in ASSCG extends beyond mere automation of coding tasks. It has fundamentally altered software engineering practices by:

1) **Accelerating Development Cycles** [39]**:** AI-driven code generation tools enable rapid prototyping and iteration, reducing time-to-market for software products.
2) **Improving Code Quality** [4]: By analyzing vast amounts of code data, AI systems can identify best practices, detect errors, and suggest improvements, leading to higher-quality code.
3) **Enhancing Developer Productivity** [3]: Developers can concentrate on higher-level design and problem-solving when repetitive coding chores are automated, which encourages creativity and innovation.
4) **Facilitating Collaboration** [40][41][42]**:** AI-driven code generation tools promote collaboration between developers, domain experts, and stakeholders by providing a common platform for expressing requirements and generating executable code.

6 Future paths and research opportunities

6.1 Emerging trends and future paths in AI innovations for ASSCG

The emerging trends and future paths in AI innovations for ASSCG hold promise for advancing the state-of-the-art in automated code generation, enhancing productivity, quality, and innovation in software engineering practices. By leveraging cutting-edge AI techniques, fostering ethical and responsible AI practices, and promoting collaboration and knowledge sharing,

ASSCG can continue to evolve and address the complex challenges of modern software development can be concluded in the Table 6:

Table 6: Emerging trends and future paths in AI innovations for ASSCG

No	Trend	Description	Implications
1	Advancements in Neural Architecture Search (NAS) [43] [44]	NAS techniques automate the design of neural network architectures, potentially revolutionizing AI-driven code generation by enabling the creation of more efficient and effective models tailored to specific tasks	NAS can lead to the development of specialized neural architectures optimized for code generation, enhancing performance, scalability, and adaptability to diverse software engineering tasks.
2	Integration of Reinforcement Learning (RL)[45]	RL algorithms enable AI systems to learn optimal decision-making strategies through trial and error, offering promising avenues for improving code generation by incorporating feedback mechanisms and adaptive learning.	Integrating RL into ASSCG can enhance the ability of AI models to generate code that meets evolving requirements, adapts to changing contexts, and optimizes for various quality metrics, such as readability, efficiency, and maintainability.
3	Hybrid Approaches Combining Symbolic and Sub symbolic AI [46]	Hybrid AI models combine symbolic reasoning with sub symbolic learning, leveraging the strengths of both approaches to enhance code generation capabilities, such as semantic understanding, context awareness, and logical inference	Integrating symbolic and sub symbolic AI techniques in ASSCG can enable more robust and interpretable code generation, addressing limitations associated with purely data-driven or rule-based approaches and supporting complex software engineering tasks.
4	Continual Learning and Lifelong Adaptation [47]	Continual learning frameworks enable AI systems to acquire and integrate new knowledge over time, facilitating lifelong adaptation to changing environments, requirements, and user feedback	Implementing continual learning mechanisms in ASSCG can enable AI models to continuously improve their code generation capabilities, learn from new codebases or programming paradigms, and adapt to emerging trends and technologies in software development.
5	Ethical and Responsible AI Practices [48]	With the increasing deployment of AI in code generation, there is a growing emphasis on ethical and responsible AI practices to address concerns related to bias, fairness, transparency, privacy, and accountability.	Building ethical considerations into ASSCG from the start is crucial for creating code that meets legal and social standards, avoids bias, and promotes responsible software development.
	Collaborative AI Development Environments [49]	Collaborative AI development environments leverage AI technologies to facilitate collaboration, code review, knowledge sharing, and collective intelligence among developers, enhancing productivity and innovation in software engineering.	Creating collaborative AI development environments tailored for ASSCG can foster interdisciplinary collaboration between AI researchers, software engineers, domain experts, and end-users, enabling the co-creation of AI models and tools that address real-world software development challenges.

6.2 Integration of AI with other software engineering methodologies

Combining AI with established methodologies like Agile, DevOps, Lean, and MDD holds immense promise for boosting software development. This integration can lead to more efficient, effective, and high-quality software. However, to fully unlock this potential, we need to address challenges like integrating the tools seamlessly, ensuring reliable and secure AI outputs, and adapting these practices to different development cultures.

1) **Agile development** [50]:

- **Description**: Agile methodologies emphasize iterative and collaborative development, focusing on delivering working software in short iterations. Integrating AI with Agile practices can enhance the efficiency and effectiveness of development processes by automating repetitive tasks, such as code generation, testing, and deployment.
- **Benefits**: AI-powered tools can automate manual tasks, provide predictive analytics for sprint planning, and optimize resource allocation based on historical data. Additionally, AI can facilitate continuous integration and delivery (CI/CD) pipelines by identifying potential bottlenecks and suggesting optimizations.
- **Challenges:** Ensuring seamless integration of AI tools with existing Agile workflows requires careful consideration of team dynamics, tool compatibility, and change management processes. Moreover, maintaining transparency and accountability in AI-

driven decision-making is essential to preserve the core principles of Agile development.

2) DevOps [50] [51]**:**

- **description**: DevOps emphasizes collaboration and automation between development and operations teams to accelerate software delivery and improve deployment frequency. Integrating AI with DevOps practices can streamline continuous integration, delivery, and deployment processes by automating code analysis, testing, and deployment tasks.
- **Benefits**: AI-powered tools can identify and prioritize software defects, optimize infrastructure provisioning, and predict system failures before they occur. Additionally, AI can analyze deployment metrics and user feedback to improve the quality and reliability of software releases.
- **Challenges**: Ensuring the reliability and security of AI-driven automation in DevOps pipelines is crucial to prevent unintended consequences and mitigate risks. Moreover, integrating AI tools with existing DevOps toolchains requires careful planning and coordination to minimize disruptions and ensure compatibility.

3) Lean software development [50]**:**

- **Description**: Lean principles focus on maximizing customer value while minimizing waste through continuous improvement and iterative delivery. Integrating AI with Lean practices can enhance software development processes by automating repetitive tasks, optimizing resource allocation, and identifying opportunities for process improvement.
- **Benefits**: AI-powered tools can analyze historical data to identify bottlenecks, predict future resource requirements, and optimize workflow efficiency. Additionally, AI can facilitate rapid prototyping and experimentation by generating code snippets or design alternatives based on user input and feedback.
- **Challenges**: Ensuring alignment between AI-driven optimizations and Lean principles, such as customer focus, waste reduction, and continuous learning, is essential to avoid conflicts and maintain the integrity of Lean practices. Moreover, fostering a culture of experimentation and continuous improvement is crucial to leverage AI effectively in Lean environments.

4) Model-Driven development (MDD) [52] [53]**:**

- **Description**: MDD emphasizes the use of high-level models to describe software systems, which are then automatically transformed into executable code. Integrating AI with MDD can enhance the expressiveness and flexibility of modeling languages, automate model transformation tasks, and improve the accuracy of code generation.
- **Benefits**: AI-powered tools can analyze natural language requirements, extract domain concepts, and automatically generate corresponding model elements. Additionally, AI can assist in model validation and verification by identifying inconsistencies, ambiguities, and missing requirements.
- **Challenges**: Ensuring the correctness and completeness of AI-generated models and code is crucial to prevent logic errors and maintain system reliability. Moreover, integrating AI tools with existing MDD tools and workflows requires standardization and interoperability to ensure seamless collaboration and compatibility.

Table 7: Integration of AI with other software engineering methodologies

Software engineering methodology	Description	Integration with AI
DevOps [50] [51]	DevOps focuses on collaboration and automation between software development and IT operations teams throughout the software development lifecycle.	AI can be integrated into DevOps practices for automated testing, continuous integration and deployment, infrastructure management, and monitoring, enhancing efficiency and reliability.
Agile [50]	Agile emphasizes iterative and incremental development, customer collaboration, and responding to change.	AI techniques can support Agile methodologies by automating repetitive tasks, analyzing user feedback, predicting project timelines and resource allocation, and optimizing sprint planning and backlog management.
Waterfall [54]	Waterfall is a sequential software development model with distinct phases such as requirements, design, implementation, testing, and maintenance.	AI can be integrated into Waterfall methodologies to automate documentation generation, requirements analysis, code generation, and testing, improving productivity and reducing manual effort.
Lean Software Development [50]	Lean focuses on minimizing waste, maximizing value, and continuously improving processes through feedback and adaptation.	AI can support Lean principles by analyzing process data, identifying bottlenecks, optimizing resource allocation, predicting project risks, and facilitating continuous improvement initiatives.
Spiral Model [54]	The Spiral model combines elements of both iterative and sequential development, with cycles of risk analysis, development, and evaluation.	AI can enhance the Spiral model by automating risk analysis, predicting project outcomes, recommending iterative refinements, and optimizing resource allocation based on evolving project requirements.
Extreme Programming (XP) [50]	XP emphasizes short development cycles, continuous testing, simplicity, and customer involvement.	AI techniques can complement XP by automating test case generation, identifying code smells and refactoring opportunities, analyzing customer feedback, and facilitating pair programming and code reviews.
Rational Unified Process (RUP)	RUP is a customizable software development process framework that provides guidelines, templates, and best practices for iterative	AI can integrate with RUP by providing decision support for process tailoring, analyzing project metrics, predicting project risks, and recommending iterative refinements based

development.	on historical data and feedback.

7 Conclusion

AI is shaking up how we write software! Developers can now automate tedious tasks, speed up development, and even improve code quality with the help of AI-powered tools. However, there are still hurdles to overcome, like making AI's decisions easier to understand, preventing bias in the code, and making these tools work well for large projects. The future of AI code generation is bright! Imagine software being built faster, with fewer bugs, and even by people who aren't professional programmers. To get there, we need to keep improving AI and figure out how to use it effectively. This will change how software is written forever, making the industry more productive, reliable, and accessible. In short, AI is bringing a new age of software development – one that's faster, smarter, and more open to everyone. By embracing AI tools, developers can solve complex coding problems and deliver top-notch software that meets the needs of today's digital world.

References

[1] Z. Bahroun, C. Anane, V. Ahmed, and A. Zacca, "Transforming Education: A Comprehensive Review of Generative Artificial Intelligence in Educational Settings through Bibliometric and Content Analysis," *Sustain.*, vol. 15, no. 17, 2023, doi: 10.3390/su151712983.

[2] F. Pinto-Santos, Z. Alizadeh-Sani, D. Alonso-Moro, A. González-Briones, P. Chamoso, and J. M. Corchado, "A Template-Based Approach to Code Generation Within an Agent Paradigm," *Commun. Comput. Inf. Sci.*, vol. 1472 CCIS, pp. 296–307, 2021, doi: 10.1007/978-3-030-85710-3_25/COVER.

[3] Ankur Desai and D. Atul, "Introducing Amazon CodeWhisperer, the ML-powered coding companion | AWS Machine Learning Blog," 2022. https://aws.amazon.com/blogs/machine-learning/introducing-amazon-codewhisperer-the-ml-powered-coding-companion/.

[4] J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro, "Automated Source Code Generation and Auto-Completion Using Deep Learning: Comparing and Discussing Current Language Model-Related Approaches," *AI 2021, Vol. 2, Pages 1-16*, vol. 2, no. 1, pp. 1–16, Jan. 2021, doi: 10.3390/AI2010001.

[5] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," *IEEE Access*, vol. 10, pp. 82434–82455, 2022, doi: 10.1109/ACCESS.2022.3196347.

[6] I. L. Yen, F. B. Bastani, F. Mohamed, H. Ma, and J. Linn, "Application of AI planning techniques to automated code synthesis and testing," *14th IEEE Int. Conf. Tools with Artif. Intell. 2002. (ICTAI 2002). Proceedings.*, pp. 131–137, 2002, doi: 10.1109/TAI.2002.1180797.

[7] J. Cruz-Benito, S. Vishwakarma, F. Martín-Fernández, I. F. I. Quantum, Electrical, and C. E. C. M. University, "Automated Source Code Generation and Auto-completion Using Deep Learning: Comparing and Discussing Current Language-Model-Related Approaches," *Appl. Informatics*, vol. 2, no. 1, pp. 1–16, Mar. 2020, doi: 10.3390/AI2010001.

[8] Y. Danilchenko and R. Fox, "Automated code generation using case-based reasoning, routine design and template-based programming," in *CEUR Workshop Proceedings*, 2012, vol. 841, pp. 119–125.

[9] H. Koziolek *et al.*, "Rule-based code generation in industrial automation: Four large-scale case studies applying the CAYENNE method," *Proc. - Int. Conf. Softw. Eng.*, pp. 152–161, Jun. 2020, doi: 10.1145/3377813.3381354.

[10] J. Zhu and M. Shen, "Research on Deep Learning Based Code Generation from Natural Language Description," *2020 IEEE 5th Int. Conf. Cloud Comput. Big Data Anal. ICCCBDA 2020*, pp. 188–193, Apr. 2020, doi: 10.1109/ICCCBDA49378.2020.9095560.

[11] O. A. C. Cortes, E. Sá, J. A. da Silva, and A. Rau-Chaplin, "An Automatic Code Generator for Parallel Evolutionary Algorithms : Achieving Speedup and Reducing the Programming Efforts," in *Conference: The Ninth International Conference on Advanced Engineering Computing and Applications in Sciences*, 2015, no. c, pp. 39–44.

[12] T. H. M. Le, H. Chen, and M. A. Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020, doi: 10.1145/3383458.

[13] X. Song, H. Sun, X. Wang, and J. Yan, "A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques," *IEEE Access*, vol. 7, pp. 111411–111428, 2019, doi: 10.1109/ACCESS.2019.2931579.

[14] H. Liao, J. Jiang, and Y. Zhang, "A Study of Automatic Code Generation," *Int. Conf. Commun. Inf. Syst.*, pp. 689–691, 2010, doi: 10.1109/ICCIS.2010.171.

[15] V. J. Bhagyashree W. Sorte, P. P. Joshi, "Use of Artificial Intelligence in Software Development Life Cycle," *J. Innov. Comput. Emerg. Technol.*, vol. 2, no. 1, 2021, doi: 10.56536/jicet.v2i1.25.

[16] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Comput. Lang. Syst. Struct.*, vol. 52, pp. 43–62, 2018, doi: 10.1016/j.cl.2017.11.003.

[17] A. T. Imam, T. Rousan, and S. Aljawarneh, "An expert code generator using rule-based and frames knowledge representation techniques," *2014 5th Int. Conf. Inf. Commun. Syst. ICICS 2014*, 2014, doi: 10.1109/IACS.2014.6841951.

[18] T. Sharma *et al.*, "A Survey on Machine Learning Techniques for Source Code Analysis," Oct. 2021, doi: 10.1145/nnnnnnn.nnnnnnn.

[19] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A Survey of Machine Learning for Big Code and Naturalness,” *ACM Comput. Surv.*, vol. 51, no. 4, Sep. 2017, doi: 10.1145/3212695.

[20] N. Beau and B. Crabbé, “The impact of lexical and grammatical processing on generating code from natural language,” *Proc. Annu. Meet. Assoc. Comput. Linguist.*, pp. 2204–2214, 2022, doi: 10.18653/V1/2022.FINDINGS-ACL.173.

[21] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, “Deep Transfer Learning for Source Code Modeling,” *https://doi.org/10.1142/S0218194020500230*, vol. 30, no. 5, pp. 649–668, Jun. 2020, doi: 10.1142/S0218194020500230.

[22] C. Liu *et al.*, “Improving ChatGPT Prompt for Code Generation,” *arXiv:2305.08360*, May 2023, Accessed: Jun. 18, 2023. [Online]. Available: http://arxiv.org/abs/2305.08360.

[23] S. Saravanan and K. Sudha, “GPT-3 Powered System for Content Generation and Transformation,” *Proc. - 2022 5th Int. Conf. Comput. Intell. Commun. Technol. CCICT 2022*, pp. 514–519, 2022, doi: 10.1109/CCICT56684.2022.00096.

[24] J. Zhou *et al.*, “Graph neural networks: A review of methods and applications,” *AI Open*, vol. 1, pp. 57–81, Jan. 2020, doi: 10.1016/J.AIOPEN.2021.01.001.

[25] V. Mironovich, M. Buzdalov, and V. Vyatkin, “Automatic generation of function block applications using evolutionary algorithms: Initial explorations,” *Proc. - 2017 IEEE 15th Int. Conf. Ind. Informatics, INDIN 2017*, pp. 700–705, Nov. 2017, doi: 10.1109/INDIN.2017.8104858.

[26] C. C. Insaurralde, “Software programmed by artificial agents toward an autonomous development process for code generation,” *Proc. - 2013 IEEE Int. Conf. Syst. Man, Cybern. SMC 2013*, pp. 3294–3299, 2013, doi: 10.1109/SMC.2013.561.

[27] J. P. A. Hoyos and F. Restrepo-Calle, “Automatic source code generation forweb-based process-oriented information systems,” in *ENASE 2017 - Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2017, pp. 103–113, doi: 10.5220/0006333901030113.

[28] D. Radošević, T. Orehovački, and I. Magdalenić, “Towards the Software Autogeneration,” *SSRN Electron. J.*, Oct. 2014, doi: 10.2139/SSRN.2505658.

[29] M. Dorin and S. Montenegro, “Metrics to understand future maintenance effort required of a complicated source code,” *Actas del Congr. Int. Ing. Sist.*, pp. 171–183, 2019, doi: 10.26439/CIIS2019.5510.

[30] P. Jackson, “Understanding understanding and ambiguity in natural language,” *Procedia Comput. Sci.*, vol. 169, pp. 209–225, Jan. 2020, doi: 10.1016/j.procs.2020.02.138.

[31] V. Saideep, “Automating Software Development using Artificial Intelligence[1] V. Saideep, ‘Automating Software Development using Artificial Intelligence,’ p. 6, Apr. 2020, doi: 10.36227/TECHRXIV.12089139.V1.,” p. 6, Apr. 2020, doi: 10.36227/TECHRXIV.12089139.V1.

[32] C. Yang, Y. Liu, and C. Yin, “Recent Advances in Intelligent Source Code Generation: A Survey on Natural Language Based Studies,” *Entropy (Basel).*, vol. 23, no. 9, Sep. 2021, doi: 10.3390/E23091174.

[33] D. Baulé, C. G. von Wangenheim, A. von Wangenheim, J. C. R. Hauck, and E. C. V. Júnior, “Automatic code generation from sketches of mobile applications in end-user development using Deep Learning,” *arxiv.org:2103.05704*, Mar. 2021, Accessed: Jun. 03, 2023. [Online]. Available: http://arxiv.org/abs/2103.05704.

[34] D. De Souza Baule, C. G. Von Wangenheim, A. Von Wangenheim, and J. C. R. Hauck, “Recent Progress in Automated Code Generation from GUI Images Using Machine Learning Techniques,” *JUCS - J. Univers. Comput. Sci. 26(9) 1095-1127*, vol. 26, no. 9, pp. 1095–1127, 2020, doi: 10.3897/JUCS.2020.058.

[35] J. Arogundade, O.T., Onilede, O., Misra, S., Abayomi-Alli, O.O., Odusami, M.O., & Oluranti, “From Modeling to Code Generation: An Enhanced and Integrated Approach,” 2021.

[36] Z. Bilgin, “Code2Image: Intelligent Code Analysis by Computer Vision Techniques and Application to Vulnerability Prediction,” *ArXiv, abs/2105.03131.*, May 2021, Accessed: Jun. 02, 2023. [Online]. Available: http://arxiv.org/abs/2105.03131.

[37] B. Yetistiren, I. Ozsoy, and E. Tuzun, “Assessing the quality of GitHub copilot’s code generation,” *PROMISE 2022 - Proc. 18th Int. Conf. Predict. Model. Data Anal. Softw. Eng. co-located with ESEC/FSE 2022*, vol. 10, no. 22, pp. 62–71, Nov. 2022, doi: 10.1145/3558489.3559072.

[38] “Google Cloud AutoML - Train models without ML expertise.” https://cloud.google.com/automl/ (accessed Jun. 18, 2023).

[39] J. G. Villalba, B. Uyanık, and A. Sayar, “Developing Web-Based Process Management with Automatic Code Generation,” *Appl. Sci. 2023, Vol. 13, Page 11737*, vol. 13, no. 21, p. 11737, Oct. 2023, doi: 10.3390/APP132111737.

[40] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration Code Generation via ChatGPT,” *ACM Trans. Softw. Eng. Methodol*, vol. 1, p. 38, Apr. 2023, Accessed: May 21, 2024. [Online]. Available: https://arxiv.org/abs/2304.07590v3.

[41] F. Wang *et al.*, “Slide4N: Creating Presentation Slides from Computational Notebooks with Human-AI Collaboration,” *Conf. Hum. Factors Comput. Syst. - Proc.*, p. 2023, Jan. 364AD, doi: 10.1145/3544548.3580753.

[42] H. Strobelt, J. Kinley, R. Krueger, J. Beyer, H. Pfister, and A. M. Rush, “GenNI: Human-AI Collaboration for Data-Backed Text Generation,” *Vol. 28, Issue 1, Pages 1106 - 1116*, vol. 28, no. 1, pp. 1106–1116, Jan. 2022, doi: 10.1109/TVCG.2021.3114845.

[43] K. T. Chitty-Venkata and A. K. Somani, "Neural Architecture Search Survey: A Hardware Perspective," *ACM Comput. Surv.*, vol. 55, no. 4, Nov. 2022, doi: 10.1145/3524500.

[44] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient Architecture Search by Network Transformation," *Proc. AAAI Conf. Artif. Intell.*, vol. 32, no. 1, pp. 2787–2794, Apr. 2018, doi: 10.1609/AAAI.V32I1.11709.

[45] M. Jones and F. Cañas, "Integrating Reinforcement Learning with Models of Representation Learning," *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, 2010. http://palm.mindmodeling.org/cogsci2010/papers/0357/paper0357.pdf (accessed Feb. 19, 2024).

[46] A. Himmelhuber, S. Grimm, M. Joblin, S. Zillner, and T. Runkler, "Combining sub-symbolic and symbolic methods for explainability," in *Frontiers in Artificial Intelligence and Applications*, vol. 369, IOS Press BV, 2023, pp. 559–576.

[47] B. Irfan, A. Ramachandran, M. Staffa, and H. Gunes, "Lifelong Learning and Personalization in Long-Term Human-Robot Interaction (LEAP-HRI): Adaptivity for All," *ACM/IEEE Int. Conf. Human-Robot Interact.*, pp. 929–931, Mar. 2023, doi: 10.1145/3568294.3579956.

[48] A. Y. A. Bani Ahmad, "Ethical implications of artificial intelligence in accounting: A framework for responsible ai adoption in multinational corporations in Jordan," *Int. J. Data Netw. Sci.*, vol. 8, no. 1, pp. 401–414, Jan. 2024, doi: 10.5267/j.ijdns.2023.9.014.

[49] R. Bharadwaj and I. Parker, "Double-edged sword of large language models: mitigating security risks of AI-generated code," in *The International Society for Optical Engineering*, 2023, p. 18, doi: 10.1117/12.2664116.

[50] A. Poth, C. Heimann, D. Eißfeldt, and S. Waschk, "Sustainable IT in an Agile DevOps Setup Leads to a Shift Left in Sustainability Engineering," in *24th International Conferences on Agile Software Development, XP 2023*, 2022, pp. 21–28, doi: 0.1007/978-3-031-48550-3_3.

[51] P. Narang and P. Mittal, "Continuous Assessment and Improvement of Software Quality with DevOps-Based Hybrid Model of Automation Tools," *J. Comput. Syst. Sci. Int.*, vol. 62, no. 2, 2023, doi: 10.1134/S1064230723020144.

[52] G. Giachetti, J. L. de la Vara, and B. Marín, "Mastering Agile Practice Adoption through a Model-Driven Approach for the Combination of Development Methods," *Vol. 65, Issue 2, Pages 103 - 125*, vol. 65, no. 2, pp. 103–125, Apr. 2023, doi: 10.1007/s12599-022-00785-5.

[53] J. Chueca, J. I. Trasobares, Á. Domingo, L. Arcega, C. Cetina, and J. Font, "Comparing software product lines and Clone and Own for game software engineering under two paradigms: Model-driven development and code-driven development," *Vol. 205*, vol. 205, Jan. 19248BC, doi: 10.1016/j.jss.2023.111824.

[54] I. Lishner and A. Shtub, "Enhancing Strategic Planning of Projects: Selecting the Right Product Development Methodology," *Inf.*, vol. 14, no. 12, 2023, doi: 10.3390/info14120632.