Use of AI-driven Code Generation Models in Teaching and Learning Programming

a Systematic Literature Review

Doga Cambaz[1]

Supervisor(s): Fenia Aivaloglou[1]**, Xiaoling Zhang**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 24, 2023

Name of the student: Doga Cambaz
Final project course: CSE3000 Research Project
Thesis committee: Fenia Aivaloglou, Xiaoling Zhang, Tom Viering

Abstract

The recent emergence of AI-driven code generation models can potentially transform programming education. To pinpoint the current state of research on using AI code generators to support learning and teaching programming, we conducted a systematic literature review with 21 papers published since 2018. The review presents the teaching and learning practices in programming education that utilize these models, the characteristics and performance indicators of the code generation models, and aspects to be considered when utilizing the models in programming education, including the risks and challenges of using code generation models for educational practices. AI code generators can be an assistive tool for both learners and instructors if the risks are mitigated.

1 Introduction

Large Language Models (LLMs), like OpenAI's Generative Pre-trained Transformer (GPT) models and the Codex model, have the ability to generate code from natural language descriptions, enabling natural language programming and performing a wide range of tasks, including code-to-code operations like code completion, translation, repair, as well as language-to-code operations like code explanation [10]. These AI-driven code generation models offer both opportunities and challenges for educators and students in programming education, as highlighted by existing research. For example, code generators might automatically correct syntax, allowing students to concentrate more on the problem-solving components of computational thinking [1]. Additionally, by producing programming exercises and explanations for the solutions, these tools could help educators develop curricula [1]. Also, another study results indicate that AI code generators allowed novice programmers to perform better and faster with less frustration, and did not reduce their performance on manual code modification or writing code in the absence of the code generator [10]. On the other hand, auto-generated code raises concerns about academic integrity and the risk of users excessively relying on the generated outputs [10].

Given their availability and accessibility, code generation models have the potential to transform how programming is taught and learned in the near future. However, the sudden emergence of these tools may catch educators off-guard, leaving them unprepared for the significant impact of code generation models on education [1]. Hence, it is critical to review and adapt our educational practices to incorporate these new technologies.

AI-driven code generation models have started to become a part of the education landscape, yet there is limited understanding of how best to adapt our teaching practices to effectively manage the challenges and benefits associated with their use. This study aims to address this gap by providing a comprehensive review of the current state of code generation models' use in teaching and learning programming, and by identifying best practices. Through this systematic literature review, we aim to synthesize guidelines that will leverage the benefits of AI-generated code while mitigating the risks.

The main question that will be answered is *How can code generation models be used in practices for teaching and learning programming?* The following research questions have been formulated to guide our systematic literature review.

- **RQ1:** What are the practices that use code generation models for teaching and learning programming?
- **RQ2:** What are the characteristics of the code generation models that are used in teaching and learning practices?
- **RQ3:** What indicators are used for evaluating the performance of code generation models in teaching and learning practices?
- **RQ4:** What aspects should be considered when utilizing code generation models in teaching and learning practices?

The rest of the paper is organized as follows: Section 2 presents the methodology, Section 3 presents the results of the review, Section 4 discusses the results and limitations, Section 5 presents threats to validity, Section 6 discusses responsible research and finally, Section 7 concludes the study.

2 Methods

We conducted a systematic literature review by adapting the guidelines proposed by Kitchenham and Charters [11]. A systematic literature review is preferred as it follows a well-defined procedure, ensuring transparency and reproducibility. Section 2.1 describes the search process, including the databases chosen and the search strings used. Section 2.2 describes the criteria for the inclusion and exclusion of the papers. Section 2.3 describes the screening process and finally, Section 2.4 explains the information extraction.

2.1 Search Process

Initially, two databases, ACM Digital Library and Scopus, were selected for relevant paper searches. ACM Digital Library covers a wide array of computer science topics, including programming languages, artificial intelligence, and human-computer interaction, making it a valuable literature source for the research questions. Secondly, Scopus is a multidisciplinary database encompassing various academic fields such as science, engineering, technology, health sciences, and social sciences. Scopus is chosen as it provides a broad range of articles from different academic fields that can help us get a comprehensive understanding of the use of code generation models in education and help us access papers that are not present in the ACM database.

The search string developed to retrieve resources from the databases is shown in Figure 1. The string was created by pinpointing search terms from the research questions, then listing their synonyms, and also by including specific large language models or code generation tools like ChatGPT and Github Copilot.

In systematic reviews, publication bias can result in systematic bias if left unaddressed [11, p. 15]. Therefore, Google

(“code generation model*” OR Codex OR “Github Copilot” OR ChatGPT OR “AI coding assistants” OR “AI code-generators” OR “code generation”) AND (teaching OR education OR learning OR educat* OR learn* OR instructor*) AND (“computer science” OR “computing education” OR programming OR “coding practices” OR “software engineering”) AND (assessment* OR curriculum OR curricula OR practices OR proposal OR tools)

Figure 1: Search String for ACM and Scopus

Scholar is used to identify additional resources. Google Scholar expands the search to include unpublished work, addressing the concern of positive outcomes being published more frequently than negative results [11, p. 15]. Furthermore, considering the recent availability of large language models and code generation models to the public, Google Scholar served as a valuable tool for identifying more resources.

Google Scholar offers limited options to combine multiple search terms with Boolean operators and does not allow you to limit your search to title, abstract, and keywords. Thus, a more strict query is used to search in the full text of publications. The new search string is provided in Figure 2.

education learning teaching Codex OR “Github Copilot” OR ChatGPT OR “AI coding assistants” OR “code generation models” “computing education” OR “programming education” curricula OR practices OR proposal OR tools OR strategies “large language models”

Figure 2: Search String for Google Scholar

The search performed on 08/05/2023 was limited to papers published in the last 5 years (2018-2023), considering the novelty of LLMs for code generation. Search results were recorded in spreadsheets, which were later merged into a *a single file* to identify duplicates and record subsequent screening.

2.2 Criteria

The following inclusion and exclusion criteria are determined to define the scope of our research.
Inclusion Criteria:

- Journal articles and conference proceedings that are written in English
- Papers that present or discuss the use of LLMs for code generation for educational purposes
- Papers published in the last five years (2018-2023)
- Papers that focus on the impact of AI code generation on Computer Science and programming education

Exclusion Criteria:

- Papers that are not written in English
- Papers irrelevant to code generation models with large language models
- Papers irrelevant to use of code generation models in programming education
- Papers that are inaccessible

2.3 Selection Process

The selection process included three steps, *Identification*, *Screening*, and *Eligibility*. Figure 3 displays each step clearly and provides the number of articles included in each step. Initially, 162 records were identified from the databases; 47 were from the ACM database and 115 were from Scopus. Furthermore, we identified 79 records from Google Scholar. Excluding duplicated results, 217 records remained for the screening process.

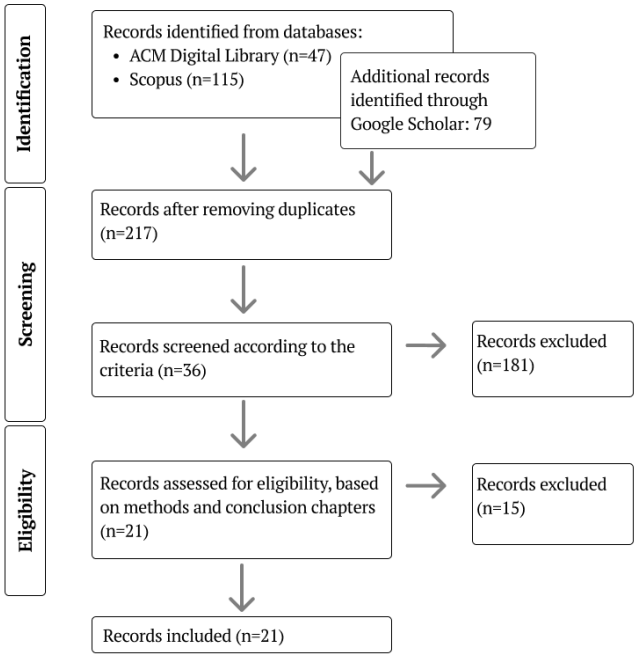

Figure 3: PRISMA flow diagram

During the Screening step, the title and abstract of each article are screened to decide whether it potentially meets the selection criteria provided in Section 2.2. As a result, 36 records were included.

During the Eligibility step, we reviewed the methodology, discussion, and conclusion sections of the articles to verify their relevance to the research scope. This step ensured that the criteria in Section 2.2 are applied correctly in the Screening step and that the selected papers address at least one sub-question, either by presenting the use of code generation models in programming education, introducing indicators to evaluate the models used, or describing aspects that should be considered when utilizing these models. After this process, we ended up with 21 relevant articles to review.

2.4 Coding and Information Extraction

Information extraction was performed using a combination of deductive and inductive coding approaches. Initially, main

themes and codes were derived deductively from the research questions, and subsequently, new codes were generated through an iterative review of the articles. Information extraction is conducted using the *Atlas.ti*[1] software.

During the reading process, extracted quotes were matched with relevant codes or themes. For RQ1, the main theme was educational practices, including teaching, learning practices, and tools. Additional codes like assessment and content generation were added during the iterative process. For RQ2, the main theme was characteristics of code generation models, with subcategories like accuracy, performance, and limitations. For RQ3, the main theme was performance indicators, after reading the papers we categorized relevant quotes as quantitative and qualitative metrics. For RQ4, we included categories of risks, ethical use, and alignment with learning objectives, which were refined upon reading the articles.

3 Results

The results of the review are analyzed for each research question. Section 3.1 presents teaching and learning practices that use LLM-based code generation models as well as the educational tools that utilize the models. Section 3.2 provides the characteristics of the code generation models used in the practices discussed in Section 3.1. Section 3.3 presents the performance metrics to evaluate the code generation models that are used in programming education. Finally, Section 3.4 discusses the aspects to be considered when using code generation models in programming education.

3.1 RQ1: The educational practices that use code generation models

The results for this research question are divided into three subcategories: teaching practices, learning practices, and educational tools that use LLM-based code generation models. Findings indicate that teachers can use the models mainly to generate assignments and evaluate student work, while for students, the models function as virtual tutors. Tables 1 and 2 list teaching and learning activities that use code generation models, with reference papers and representative quotes.

Teaching Practices

Instructors can utilize LLM-based code generation models to automate the generation of programming assignments, including sample answers with explanations and test cases. These models also facilitate code translation between programming languages [3], simplifying content creation. By leveraging these models, instructors can also generate novel exercise variations based on existing exercises [8]. Moreover, the models' ability to contextualize problem statements [19] and create personalized questions tailored to student's interests [13] allows instructors to create engaging questions. It is worth mentioning that the papers in Table 1 all recognize the necessity of manually reviewing AI-generated materials to ensure accuracy and clarity, given the unreliability of LLMs. However, researchers still agree that this practice reduces teachers' workload, as evaluation is easier compared to creating exercises from scratch. For instance, Sarsa et al. [19] state that code generation tools aid instructors in overcoming writer's block and generating ideas quickly, even if the resulting exercises are not used directly. Moreover, Geng et al. [8] suggest that instructors and TAs can manually review the generated exercises or homework to ensure correctness and appropriateness.

Furthermore, LLMs can assist teachers in partially automating the grading process by identifying strengths and weaknesses in writing assignments, and providing natural language feedback on student code, as presented in Table 1. This capability can save teachers considerable time when providing individualized feedback to students.

Learning Practices

According to the papers, LLM-based code generation models can be used by learners to generate practice materials, alternative solutions for programming questions, code explanations, and suggestions. Table 2 summarizes the learner activities. Overall, the models can help students study and practice programming by generating personalized learning materials and functioning as a tutor, especially for students who do not have access to tutoring.

Code generation models can create additional learning resources in programming education, such as practice exercises, sample answers, and alternative solutions. According to Geng et al. [8], these models can generate personalized exercises tailored to the student's proficiency level, improving the learning experience. The models can also offer multiple solutions to a given programming problem, introducing students to different problem-solving approaches. Furthermore, tools like Github Copilot and ChatGPT offer valuable assistance to learners by providing immediate feedback on their code. They can explain the code, suggest optimizations, provide syntax tips, clarify error messages, and suggest ways to fix the errors. Moreover, these tools are capable of various types of code explanations. Given a code snippet, GPT-3 can analyze time complexity, identify common mistakes, summarize code, trace execution, fix bugs and explain how they were fixed, create real-world analogies, list relevant programming concepts, and predict console output [14].

Moreover, the added benefit of ChatGPT, mentioned in [18, 9], is its ability to generate conversational dialogues, allowing learners to ask questions in the same way they would ask their tutors, thus making the learning process more intuitive, interactive, and beginner-friendly. Additionally, [8] highlights that since students can input their own text prompts, these models help them learn at their preferred pace and in alignment with their learning style.

Educational Tools

Three papers [23, 5, 10] extensively explain and evaluate an educational tool that uses LLM-based code generation models. This subsection explores the *Robosourcing model* [5], *The Coding Steps* web app for learning basic Python programming [10], and the *MMAPR model* [23] for repairing bugs in student code.

The Coding Steps [10] is a web app designed for learning basic Python programming. It offers a beginner-friendly programming environment with a series of tasks introducing new concepts. Learners can progress through these tasks

[1] https://atlasti.com/

Activity	Activity Detail	Representative Quotes	References
Automatic generation of assignments	Exercise descriptions, sample answers and explanations	"[...] programming exercise, its solution, and the code explanation were all generated automatically by OpenAI Codex" [19] "ChatGPT can help instructors generate exercises, quizzes, and scenarios for student assessment" [12]	[13, 22, 9, 12, 19]
	Test cases for the exercises	"OpenAI Codex model is able to provide [...] automated tests to verify the student's solutions, and additional code explanations." [9]	[3, 9, 19]
	Personalized problems	"generate a personalized Parsons problem, one that is based on the student's incorrect solution using LLMs" [3]	[13, 3, 19]
	Variations of questions	"By simply inputting a prompt or a topic, ChatGPT can generate a range of questions with varying levels of difficulty and complexity." [8] "Codex can generate novel variations of the exercise that could then be deployed to students" [19]	[8, 19]
Assessment and evaluation	Grading Assignment	"Teachers can use large language models to semi-automate the grading of student work by highlighting potential strengths and weakness of the work in question" [9] "the model can be used to grade assignments and quizzes" [18]	[17, 18, 9]
	Identifying areas students are struggling	"[...] can help teachers to identify areas where students are struggling, which adds to more accurate assessments of student learning development and challenges." [9]	[9]
	Feedback Generation	"[...] used NLP-based models to generate feedback for textual student answers in large courses, where grading effort could be reduced by up to 85 % with a high precision and an improved quality perceived by the students." [9]	[18, 22, 9, 19]

Table 1: Teaching practices using Code Generation Models

Activity	Representative Quotes	References
Generate practise exercises	"ChatGPT can be utilized to generate exercises that are customized to a student's individual skill level, allowing them to practice and apply programming concepts in a targeted and effective manner." [8] "Our results suggest that it may be possible for individual students to provide their own keywords and have tailored exercises generated for their personal use, possibly using teacher created exercises as primes." [19]	[18, 8, 9, 12, 19]
Generate exemplar solutions	"AI-generated solutions provide a low-cost way for students to generate exemplar solutions to check their work when practicing" [1] "For example, as model solutions have been proposed as a support mechanism in introductory programming Nygren et al. (2019), students could generate model solutions with Codex for historical assignment, test and exam problems, where solutions may not otherwise exist." [19]	[18, 12, 1, 2, 19]
Generate alternative solutions	"These models could suggest alternative, and more efficient or idiomatic ways of implementing programs, which could help learners to improve their coding style." [1] "Code generation tools can also be used to help expose students to the variety of ways that a problem can be solved." [1] "They could also generate alternative correct solutions for a problem they have solved, to reflect on their own solution and to compare different algorithms and language constructs." [19]	[8, 15, 1, 19]
Improve student code	"ChatGPT can help optimize codes by suggesting ways to reduce the memory usage and time complexity" [18] "ChatGPT can identify errors in code and provide potential suggestions and code snippets."[18]	[18, 23]
Clarify error messages and provide suggestions	"For example, if a student is working with a programming language that they are not familiar with, ChatGPT can provide assistance by clarifying error messages and offering suggestions for how to fix them." [8] "Codex is capable of explaining error messages in natural language - often effectively - and that that it can also provide correct fixes based on input code and error messages" [1]	[13, 8, 1]
Support conceptual understanding	"ChatGPT can generate easy-to-understand explanations and pseudocode that are useful for learners to understand algorithmic concepts." [18]	[13, 17, 8, 18, 20, 1]
Provide syntax tips	"ChatGPT can also provide syntax tips to help students write correct and efficient code. This includes suggestions for common programming structures and best practices, as well as guidance on how to avoid syntax errors." [8]	[8, 15]
Code explanations	"GPT models are capable of explaining code in plain and easily understandable terms." [20] "This use case shows GPT-3 can identify and explain time complexity." [14] "GPT-3 can automatically create a checklist of common mistakes students might make regarding a given code snippet." [14]	[13, 17, 3, 1, 20, 14]

Table 2: Learning practices using Code Generation Models

independently while having access to an AI code generator powered by OpenAI Codex. To enable AI code generation, users input their desired code behavior in natural language using a textbox, and clicking the generate button inserts the code generated by OpenAI Codex. The prompt message for each API call to Codex is customized by combining six predefined examples, existing code in the editor, and the user's requested behavior. This conditioning ensures the AI model generates beginner-level Python code that considers the user's context. Furthermore, the study examining novice programmers' interaction with 'The Coding Steps' demonstrated that students who utilized the AI generator were able to understand and work with the generated code effectively. Using an AI code generator improved task completion and correctness scores while reducing errors and completion times. Additionally, these students performed similarly to the control group in manual code modification tasks, suggesting no decline in manual-coding performance.

Robosourcing [5] is a model for generating practice questions in a scalable manner as a learnersourcing activity. Learnersourcing involves learners collectively creating content for future learners while enhancing their own learning experience [21]. In the Robosourcing model, learners first provide a priming exercise that includes a problem statement, sample solution, and relevant themes. Using this, the system generates a pool of exercises that undergo initial automatic filtering. Learners can further filter and edit the exercises before adding them to an exercise database. This model utilizes AI code generation to shift the learner's focus from content creation to evaluation. Also, the researchers tested the feasibility of the Robosourcing model using Codex and found impressive outcomes in generating coherent programming exercises with sample solutions and automated tests. Although there were some accuracy concerns, these issues could be easily fixed manually.

Finally, Zhang et al. [23] presented *MMAPR*, a multi-modal automated repair system that utilizes Codex as its core component to automatically fix errors in students' Python programming assignments. It considers multiple sources that aid in code repair, including instructor-provided test cases, task descriptions in natural language, compiler messages for syntax errors, and even other students' submissions if accessible. The system also aims to minimize unnecessary code generation and avoids modifying correct portions of the program. MMAPR, utilizing Codex as its core, can address both syntactic and semantic errors in Python assignments through a combination of methods like multi-modal prompts, iterative querying and program chunkung.

3.2 RQ2: Characteristics of the Code Generation Models used in Educational Practises

All reviewed papers employed OpenAI's various GPT models for code generation. Among them, six studies [23, 5, 19, 7, 10, 2] used OpenAI Codex, while three studies [13, 15, 16] focused on analyzing Github Copilot, which is powered by Codex too. One study [14] used GPT-3, while [20] studied both GPT-3 and GPT-3.5. Several studies [17, 6, 8, 18, 9, 12] explored the use of ChatGPT on various educational tasks, while only one [6] used its latest version, GPT-4. Lastly, [3, 4, 22, 1] did not specify an LLM, instead focusing on the educational opportunities and challenges of AI code generation.

Performance and Limitations of the Models

Out of the 21 papers reviewed, 10 [17, 6, 8, 18, 23, 5, 20, 19, 7, 16] feature empirical studies examining the performance and characteristics of code generation models. Among these, 6 papers [17, 6, 8, 18, 20, 7, 16] argue that while the models can generally produce well-structured and accurate solutions for programming assignments, they have several limitations. Furthermore, [5, 19] discuss the limitations of Codex in generating programming assignments, while [19] also delves into the characteristics of generated code explanations.

The study in [17] involving 24 students revealed that while students who used ChatGPT achieved higher scores in less time, they encountered inaccuracies or inconsistencies in the generated code, preventing perfect scores. The study notes that ChatGPT provides coherent and well-structured code with comprehensive explanations, but executing the generated code in the IDE can lead to errors due to limitations in input/output size, output truncation, and inconsistencies resulting from multiple queries to ChatGPT. Similarly, in another study [18], researchers found that adjustments to the generated code were still required for error-free compilation, despite ChatGPT achieving an average code accuracy of 85.42% when generating code from problem descriptions. Moreover, Codex's performance is similar to ChatGPT. By evaluating Codex's responses to 23 programming questions from their CS1 course, Finnie-Ansley et al. [7] observed that 10 of them were successfully solved on the first attempt, some with minor formatting errors. Codex achieved a rank of 17 out of 71 students, placing it in the top quartile of class performance. Similarly, [16] indicated that some of Github Copilot's code solutions lacked clarity, for instance, they included redundant opening and closing files and unreachable code.

Furthermore, Savelka et al. [20] evaluated GPT models on 530 multiple-choice questions (MCQ) from three Python courses and found out that GPT models perform better in handling questions involving the generation of code or natural language explanations compared to MCQs. They further demonstrated that GPT models perform worse on MCQs with code snippets than those without. While fill-in-the-blank questions and completing natural language statements about code are handled relatively well, MCQs requiring analysis and reasoning about code, such as true/false questions or predicting output, are the most challenging. Similarly, Dobslaw and Bergh [6] supported these findings, showing that even the latest GPT model, GPT-4, faces challenges with various question types. GPT-4 frequently struggled with MCQs, often selecting only some of the correct options. Additionally, it had difficulties in questions involving graph traversal and executing search algorithms based solely on textual descriptions of graphs.

Regarding the generation of programming exercises, Denny et al. [5] observed that approximately one-third of the exercises generated by Codex were deemed immediately usable for teaching purposes and served as starting points for learners to evaluate and modify. Likewise, Sarsa et al. [19] also noted that Codex-generated programming exercises of-

ten required adjustments before using them in a course, as problem statements frequently did not address corner cases, and many exercises either lacked tests or had flawed ones.

Finally, regarding code explanations, [19] found that Codex explanations cover approximately 90% of the code but contain inaccuracies in about 67.2% of the explanation lines. However, these errors are usually minor and can be easily addressed by instructors or teaching assistants. More importantly, the study highlights the limitations of Codex in generating high-level code descriptions. Despite attempting various explicit priming statements, Codex tends to generate line-by-line code explanations.

3.3 RQ3: Indicators for Evaluating the Performance of Code generation models in teaching and learning practices

Three papers [5, 19, 10] systematically analyze the performance of the models with a list of performance indicators. To study the performance of LLM-based code generation models for automatically generating exercises, sample solutions, and code explanations, Denny et al. [5] and Sarsa et al. [19] used a list of performance metrics which is provided in Table 3. On the other hand, Kazemitabaar et al. [10] focused on evaluating the impact of AI code generators on learner behavior rather than directly examining the performance of the models.

The metrics used in [10] are listed with their definitions in Appendix A.1. The metrics were categorized into three groups: (i) overall training metrics which include indicators like completion rate and the amount of received feedback, (ii) per-task performance which involves correctness score, completion time, and encountered errors, and (iii) AI code generator usage that includes metrics such as the percentage of code written by Codex, and the Jaccard text similarity between final submission and Codex-generated code.

3.4 RQ4: Aspects to be considered when using code generation models in teaching and learning

Academic Integrity Eleven papers [17, 8, 15, 18, 5, 20, 9, 12, 1, 7, 16] discuss the risk of AI code generators facilitating plagiarism and compromising academic integrity. Denny et al. [5] states that familiarizing students with these technologies by using them in education could increase their use for plagiarism. Moreover, [15, 18, 7] mention the difficulty in detecting AI-generated answers. AI-generated code is diverse in structure, resistant to standard plagiarism detection tools, and does not involve communication with others like traditional cheating methods, making it difficult to detect misconduct. On top of this, Geng et al. [8] highlight that if these tools are used for assessment purposes, it may undermine the validity of academic assessments. To mitigate the risks, Qureshi [17] suggests revising academic integrity policies and honor codes to address the use of AI tools, providing clear and simple guidelines for the proper use of LLMs in education, and training students on academic integrity to ensure they fully understand the importance of maintaining ethical standards. [9] adds to the mitigation strategy by encouraging research on analysis techniques and measures to distinguish machine-generated from human-generated text, and incentives to develop curricula and instructions that require the creative and complementary use of code generation models.

Over-reliance Eight papers [17, 8, 15, 18, 20, 1, 2, 9] highlight the risk of over-reliance on code generation tools. Rahman and Watanobe [18] and Kasneci et al. [9] accentuated that the ease of acquiring answers and code from these tools can be a barrier to improving learners' critical thinking and problem-solving skills. Over-reliance on the models can lead to the loss of creativity [17], and amplify laziness [18] as the AI-generated responses may discourage students from exploring alternative solutions. Likewise, Prather et al. [15] hypothesized that "over-reliance on tools like Copilot could possibly worsen a novice's metacognitive programming skills and behaviors". Moreover, Becker et al. [1] stated that novices using models like Github Copilot, which provide embedded support in an IDE, may become reliant on auto-suggested solutions, potentially resulting in students not reading problem statements carefully and a lack of critical thinking about the steps required to solve a problem. Geng et al. [8] argues that this dependency on AI-generated code could gradually diminish the quality of education and devalue computer science degrees. Rahman and Watanobe [18] adds onto that by claiming additional research is needed to develop academic curricula, question-and-answer formats, and exams that effectively tackle the raised challenges.

Accuracy and Reliability of the models Accuracy and reliability concerns of Codex and ChatGPT are highlighted in five papers [17, 6, 9, 12, 1]. The lack of consistency and accuracy in the code generation models' responses are already mentioned in Section 3.3. Likewise, according to [17], ChatGPT has a tendency to generate solutions with non-existent rules or equations and to provide unreliable, untraceable, and unverifiable answers. It is also noted in [6] that ChatGPT tends to make errors in arithmetic and deduction while supporting them with excellent explanations. This ability of the models makes it difficult for students to distinguish the errors and unverified information, leading to students accepting false or misleading information as true [9]. Likewise, according to [1], Codex can recommend syntactically incorrect code, including undefined variables, functions, and attributes. The suggested solutions from Codex may seem correct at first glance but may not actually fulfill the intended task. To mitigate the risks, [9] stresses educating students on the critical evaluation of information and teaching strategies for exploration, investigation, and verification.

Appropriateness for beginners Papers [9, 1, 7] raise concerns about the appropriateness of these code generation models for beginners. According to [7], students using Codex to generate model solutions for exercises may hinder their learning if the generated solutions are incorrect or of poor style, resulting in the adoption of inappropriate conventions and poor coding style. While this is a risk with any crowdsourced solution, the customized nature of Codex's solutions may lead students to perceive them as more credible. Furthermore, [1] states that the coding styles of publicly available code are different, and potentially more advanced, compared

Qualitative Metrics for Programming Exercises	Definition
Sensibleness	whether the problem statement describes a practical problem that could be given to students to solve [5, 19]
Novelty	whether the copy of the programming exercise or a similar programming exercise already exists and can be found online [5, 19]
Topicality	whether the generated problem incorporates the provided theme and concepts from the required sets (e.g. matches the CS concepts provided in the prompt) [5]
Readiness for Use	the amount of manual work a teacher would have to make for the exercises and the associated sample solution and tests [5, 19]
Quantitative Indicators for Programming Exercises	**Definition**
Executability of Sample Solutions	whether the sample solutions could be run [5, 19]
Automated Tests	whether the sample solution passed the automated tests [5, 19]
Statement Coverage	the statement coverage of the automated tests when the code runs [5, 19]
Metrics for Code Explanations	**Definition**
Presence and Frequency of Mistakes	the types of mistakes present and determining and how common they were in the explanations for the different priming programs [19]
Completeness of Code Explanations	whether all parts of the code were explained in the generated explanations (Yes/No) [19]
Accuracy of Explained Lines	the proportion of correctly explained lines out of all the generated explanation lines, indicating the accuracy of the code explanations [19]

Table 3: Metrics used for evaluating automatically generated programming exercises and natural language explanations of code samples. The qualitative metrics were first assessed by Yes / No / Maybe statements by the researchers and then were quantitatively analyzed by Yes / No / Maybe counts. The quantitative metrics for evaluating the automatically generated programming exercises are conducted programmatically.

to those of typical novice programmers. Given that these models are trained on publicly available code, the style of the generated code may differ from those of typical novice programmers and their instructors.

Other ethical implications Papers also discussed harmful biases and the issue of code reuse. Five papers [15, 18, 5, 9, 1] highlight the issue of harmful bias, claiming that code generation models are not immune to the bias in AI, and can possibly reflect stereotypes, represent only certain groups of people, etc. Furthermore, two papers [15, 1] highlight that the AI-generated code can present challenges in terms of licensing and attribution. Publicly available codes that are used to train these models may have various licenses. However, AI-generated code often lacks clear attribution, leading to potential license violations. Thus, instructors should educate their students about how the models are trained and their responsibilities when reusing code.

The Future of Programming Education with Code Generation Models

Two papers [6, 7] propose contextualized, specific, and applied assessments that enable students to utilize code generation tools while still engaging in problem-solving. Likewise, [4] advocates shifting the focus from detecting and preventing the use of these tools to embracing and integrating them.

Furthermore, Becker et al. [1] and MacNeil et al. [13] suggested a shift in the focus of programming courses. While teaching in CS1 traditionally focuses on syntax and basic programming principles, as code generation models can handle low-level implementation tasks, students can shift their focus to higher-level algorithms. This could lead to a teaching approach that prioritizes algorithmic problem-solving, utilizing AI code generation for implementation and delaying syntax discussions until later stages [1]. Similarly, according to [13], as software engineers may take on more design-oriented roles in the future, the focus of courses could potentially shift towards prompt engineering, code evaluation, and debugging.

Finally, Dobslaw and Bergh [6] and Geng et al. [8] discuss the need for a transitional period for novice programmers. As mentioned earlier, understanding and assessing AI-generated code may be challenging for new programmers. Thus, programming education is still crucial for individuals to effectively use code generation tools [8]. Instead of introducing students to these technologies from day one, it may be more beneficial for them to prioritize building a strong foundation in core computing concepts [6].

4 Discussion

The findings suggested that LLM-based code generators have the potential to improve teaching and learning by acting as a teacher's assistant and a student's virtual tutor. However, their performance is not always satisfactory, and the risks of using them should be considered at all times.

For RQ1, the teaching and learning practices found in the papers were generally repetitive and lacked depth, with many papers citing the same research. The proposed learning and teaching activities were often shallow, such as 'personalized learning', and not always backed by empirical studies. This emphasizes the need for further research on the practical applications of code generation models in computing education, as there are still significant knowledge gaps and uncertainties among educators regarding the effective integration of these models into teaching and learning processes. Nevertheless,

it is still clear that these tools are only assistive and cannot replace a teacher. Furthermore, the evaluated prototypes and tools provide a starting point to utilize code generation models effectively in programming education. Particularly, the Coding Steps web app [10] reveals how AI coding assistants developed by education professionals can support complete beginners while incorporating control mechanisms to prevent over-utilization. In light of this study, we believe that especially for novice programmers, prompts can be tailored to limit the generation of huge chunks of code and constraints can be imposed to prevent direct use of generated code.

Answering RQ2 proved challenging due to the diversity of code generation models examined in the papers and their varying versions, making it difficult to generalize their characteristics and limitations. Additionally, the empirical studies differed significantly in their experimental designs, ranging from large-scale evaluations using hundreds of programming questions to smaller-scale evaluations involving twenty students. Nevertheless, we concluded that various versions of GPT-3, GPT-3.5, and Codex share similar limitations. While they excel at generating code and text explanations, they struggle with reasoning-based questions such as MCQs. We believe it is crucial to clearly communicate the limitations and unreliability of these tools to students so that they do not consider the generated answers as the hard truth.

In response to RQ3, while evaluating the accuracy of the generated material is crucial, the qualitative metrics proposed by [8, 10] - sensibleness, novelty, topicality, and readiness for use - offer valuable insights for other researchers and educators to evaluate the AI-generated code and teaching material.

Regarding RQ4, the reviewed papers discussed the challenges of using models in education but lacked specific actionable guidelines for educators to ensure safe student interaction. Further research in developing new technologies, such as ways to obstruct over-utilization or AI-based plagiarism detectors to safeguard the integrity of education, is necessary. Furthermore, we suggest developing novice-friendly user interfaces and new tools to promote the safe and appropriate use of code generation models in education, by addressing academic integrity, over-reliance, and unreliability concerns.

Finally, it should be noted that the issue of unequal access to education persists due to the dominance of English-based innovations. 4 out of 21 reviewed papers [22, 9, 19, 10] address the research and innovation gap in this area for non-English languages. While LLMs hold promise for enhancing programming education in English, they can lead to unfair access to educational technologies for non-English speakers. Unfortunately, the impact of code generation models on non-English programming education remains unexplored.

5 Threats to Validity

The present study faces several threats to its validity due to limitations in its design and execution. Firstly, the study was conducted within a relatively short time frame of 9 weeks, only by one person. This limited time may have constrained the researcher's ability to thoroughly review a wide range of relevant papers and fully delve into the topic. Additionally, as systematic literature reviews typically benefit from the expertise and input of multiple researchers, the absence of a collaborative effort may have further restricted the scope and depth of the study.

Additionally, the research area is relatively new, indicating that more extensive research is likely to emerge in the future. Consequently, the findings may be omitting important studies or advancements that were published during or after the study was conducted.

Another potential threat to the validity of this study is the dynamic nature of LLMs, with continuous improvements to the models. Since the papers reviewed in this study presented results based on current and older versions of GPT models, it is important to acknowledge that the empirical results reported in these papers might differ if evaluated on updated versions of the models, which might alter the recommendations that were presented on the use of these models. Consequently, the findings of this study may not hold for future versions of the models.

Another limitation stems from our decision to include only literature written in English. This language restriction was driven by the author's comfort and the availability of a greater number of papers in English. However, this choice could have resulted in the exclusion of relevant studies written in other languages, potentially introducing bias and limiting the generalisability of the findings.

Furthermore, it should be noted that the author does not possess expertise in programming education or LLMs. This lack of specialized knowledge may have influenced the depth of the analysis.

Despite these limitations, it is hoped that this study will contribute to a better understanding of the use of code generation models in programming education and stimulate further research in this field.

6 Responsible Research

To ensure reproducibility, the systematic literature review is chosen as the method of the study, as systematic literature reviews have a solid search method. We clearly stated how the data is obtained and provided the list of papers obtained in each step as a spreadsheet. Moreover, to mitigate the potential impact of publication bias, which can skew research outcomes, we used Google Scholar as an additional resource alongside two scientific databases to include both published and unpublished works. Additionally, to avoid confirmation bias, which is our tendency to search for information that supports our beliefs and interpret evidence in ways that are preferential to our existing beliefs, we went through all the presented data in a systematic way to avoid jumping to conclusions. We first extracted quotes and assigned them to codes, then we tried to categorize the quotes again to put them into tables or paragraphs. We went back to the papers multiple times to ensure we are not skipping any data. This approach helped mitigate the impact of personal beliefs, leading to more reliable and unbiased research outcomes.

Finally, we acknowledge that due to our decision to include sources from Google Scholar, some of the included papers in this study are preprints and not peer-reviewed, which raises questions about the quality of their findings. Therefore, fur-

ther systematic reviews should be conducted once more peer-reviewed articles on LLM-based code generation models are available.

7 Conclusion

The use of LLM-based code generators in programming education presents a promising avenue with possibilities to improve student's learning experience and alleviate the workload of teachers by providing assistance. However, in order to fully utilize their educational potential, it is essential to thoroughly analyze the limitations and risks associated with the use of these tools.

This systematic review of 21 articles emphasized the use of AI code generators in programming education and its potential advantages as a teacher's assistant in the creation and evaluation of assessments and as a student's virtual tutor that creates practice material, offers feedback, and provides suggestions. While the study reflects optimism regarding the opportunities presented by code generation tools for education, we also synthesized the limitations and risks of using them in programming education, while offering insights into the future of programming education.

While code generation models have transformative potential in teaching and learning, ensuring safe usage is crucial as failure to address the models' accuracy limitations, risk of misconduct and over-reliance pose a significant danger to computing education. Future studies should explore integrating AI code generators in classrooms and designing programming assessments that encourage critical thinking rather than relying on these tools as answer generators.

References

[1] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. "Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation". In: Association for Computing Machinery, 2023, pp. 500–506. ISBN: 9781450394314. DOI: 10.1145/3545945.3569759. URL: https://doi.org/10.1145/3545945.3569759.

[2] Robert W. Brennan and Jonathan Lesage. *Exploring the Implications of OpenAI Codex on Education for Industry 4.0*. 2023. DOI: 10.1007/978-3-031-24291-5_20.

[3] Peter Brusilovsky, Barbara J. Ericson, Christian Servin, Frank Vahid, and Craig Zilles. "The Future of Computing Education Materials". In: CS2023: ACM/IEEE-CS/AAAI Computer Science Curricula, Curricula Practices, 2023.

[4] Christopher Bull and Ahmed Kharrufa. *Generative AI Assistants in Software Development Education*. 2023. arXiv: 2303.13936 [cs.SE].

[5] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. *Robosourcing Educational Resources – Leveraging Large Language Models for Learnersourcing*. Nov. 2022. arXiv: 2211.04715 [cs.HC].

[6] Felix Dobslaw and Peter Bergh. *Experiences with Remote Examination Formats in Light of GPT-4*. 2023. arXiv: 2305.02198 [cs.CY].

[7] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. "The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming". In: Association for Computing Machinery, 2022, pp. 10–19. ISBN: 9781450396431. DOI: 10.1145/3511861.3511863. URL: https://doi.org/10.1145/3511861.3511863.

[8] Chuqin Geng, Yihan Zhang, Brigitte Pientka, and Xujie Si. *Can ChatGPT Pass An Introductory Level Functional Language Programming Course?* Apr. 2023. arXiv: 2305.02230 [cs.CY].

[9] E. Kasneci, K. Sessler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günnemann, E. Hüllermeier, J. Kuhn, and G. Kasneci. "ChatGPT for good? On opportunities and challenges of large language models for education". In: *Learning and Individual Differences* 103 (2023). DOI: 10.1016/j.lindif.2023.102274.

[10] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. "Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming". In: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3580919. URL: https://doi.org/10.1145/3544548.3580919.

[11] B Kitchenham and S Charters. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. EBSE-2007-01. Keele University and Durham University, 2007.

[12] Chung Kwan Lo. "What Is the Impact of ChatGPT on Education? A Rapid Review of the Literature". In: *Education Sciences* 13 (4 Apr. 2023), p. 410. ISSN: 2227-7102. DOI: 10.3390/educsci13040410.

[13] Stephen MacNeil, Andrew Tran, Juho Leinonen, Paul Denny, Joanne Kim, Arto Hellas, Seth Bernstein, and Sami Sarsa. "Automatically Generating CS Learning Materials with Large Language Models". In: ACM, Mar. 2022, pp. 1176–1176. ISBN: 9781450394338. DOI: 10.1145/3545947.3569630.

[14] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. "Generating Diverse Code Explanations using the GPT-3 Large Language Model". In: ACM, Aug. 2022, pp. 37–39. ISBN: 9781450391955. DOI: 10.1145/3501709.3544280.

[15] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. *"It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers*. 2023. arXiv: 2304.02491 [cs.HC].

[16] Ben Puryear and Gina Sprint. “Github Copilot in the Classroom: Learning to Code with AI Assistance”. In: *J. Comput. Sci. Coll.* 38 (1 Nov. 2022), pp. 37–47. ISSN: 1937-4771.

[17] Basit Qureshi. *Exploring the Use of ChatGPT as a Tool for Learning and Assessment in Undergraduate Computer Science Curriculum: Opportunities and Challenges*. Apr. 2023. arXiv: 2304.11214 [cs.CY].

[18] Md. Mostafizer Rahman and Yutaka Watanobe. “ChatGPT for Education and Research: Opportunities, Threats, and Strategies”. In: *Applied Sciences* 13 (9 2023). ISSN: 2076-3417. DOI: 10.3390/app13095783. URL: https://www.mdpi.com/2076-3417/13/9/5783.

[19] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. “Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models”. In: *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*. ICER ’22. Lugano and Virtual Event, Switzerland: Association for Computing Machinery, 2022, pp. 27–43. ISBN: 9781450391948. DOI: 10.1145/3501385.3543957. URL: https://doi.org/10.1145/3501385.3543957.

[20] Jaromir Savelka, Arav Agarwal, Christopher Bogart, and Majd Sakr. *Large Language Models (GPT) Struggle to Answer Multiple-Choice Questions about Code*. Mar. 2023. arXiv: 2303.08033 [cs.CL].

[21] Anjali Singh, Christopher Brooks, and Shayan Doroudi. “Learnersourcing in Theory and Practice: Synthesizing the Literature and Charting the Future”. In: *Proceedings of the Ninth ACM Conference on Learning @ Scale*. L@S ’22. New York City, NY, USA: Association for Computing Machinery, 2022, pp. 234–245. ISBN: 9781450391580. DOI: 10.1145/3491140.3528277. URL: https://doi.org/10.1145/3491140.3528277.

[22] Lixiang Yan, Lele Sha, Linxuan Zhao, Yuheng Li, Roberto Martinez-Maldonado, Guanliang Chen, Xinyu Li, Yueqiao Jin, and Dragan Gašević. *Practical and Ethical Challenges of Large Language Models in Education: A Systematic Literature Review*. Mar. 2023. arXiv: 2303.13379 [cs.CL].

[23] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. *Repairing Bugs in Python Assignments Using Large Language Models*. 2022. arXiv: 2209.14876 [cs.SE].

A Appendices

A.1 Metrics used to analyze learners' performance and behavior in [10]

Overall Training Metrics	**Definition and Source**
Completion Rate (percentage)	*Definition*: How far a learner progressed through the training phase regardless of correctness of tasks or skips: number of seen tasks divided by total tasks count.
Personalized Feedback (count)	*Definition*: Total number of personalized feedbacks a learner received during the training phase. *Source*: code submission logs.
Feedback length (characters)	*Definition*: The length (number of characters) of the personalized feedback a learner received during the training phase. *Source*: code submission logs.
Direct hints (count)	*Definition*: Total number of personalized feedbacks a learner received that included direct hints towards solving the problem. *Source*: code submission logs.
Per-Task Performance	**Definition and Source**
Coding Correctness Score (percentage)	*Definition*: How correct was a learner's solution to a single task. *Source*: Final submission in the submissions log that was graded independently by two researchers.
MCQ Correctness Score (percentage)	*Definition*: Whether a learner responded correctly to a multiple-choice question. *Source*: submission logs.
Completion Time (seconds)	*Definition*: Active time a learner spent working on a task (by removing inactivity gaps of longer than one minute). *Source*: aggregated logs.
Documentation Referenced (count)	*Definition*: Whether a learner referenced the Python documentation for a task or not. *Source*: documentation logs.
Encountered Errors (count)	*Definition*: Number of errors a learner encountered after running their code categorized into syntax, data-type, and semantic errors. *Source*: console logs.
AI Code Generator Usage	**Definition and Source**
Code Generator Usage Per Task (count)	*Definition*: Number of unique prompts and codes generated using the AI code generator during a single task. *Source*: code generator logs.
AI-Generated Code Ratio (percentage)	*Definition*: The percentage of code in a task that was generated by an AI code generator, as opposed to being written manually by the learner calculated using the Jaccard text similarity coefficient [37]. *Source*: code submission logs and code generator logs.
Tasks Broke Down into Subgoals (count)	*Definition*: Whether different parts of the final submission for a task was generated from different codex usages. *Calculation Method*: averaging over the maximum hamming distance between each line of the final submission and each line in the codex generated codes. *Source*: code submission logs and code generator logs.
Prompt Similarity with Task Description (percentage)	*Definition*: Similarity between the prompt used for generating code and the task description. *Source*: code generator logs and task descriptions.

Table 4: Definitions, sources, and calculation methods of metrics used for evaluating learner behavior as they interacted with *The Coding Steps* web app [10]