TYPE Original Research
PUBLISHED 23 May 2025
DOI 10.3389/fcomp.2025.1549761

OPEN ACCESS

EDITED BY
Tisni Santika,
Universitas Pasundan, Indonesia

REVIEWED BY
Abdallah Qusef,
Princess Sumaya University for Technology, Jordan
Reyes Juárez-Ramírez,
Universidad Autónoma de Baja California, Tijuana, Mexico

*CORRESPONDENCE
Shariq Bashir
sbmirza@imamu.edu.sa

RECEIVED 21 December 2024
ACCEPTED 29 April 2025
PUBLISHED 23 May 2025

CITATION
Bashir S (2025) Using pseudo-AI submissions for detecting AI-generated code. *Front. Comput. Sci.* 7:1549761.
doi: 10.3389/fcomp.2025.1549761

Using pseudo-AI submissions for detecting AI-generated code

Shariq Bashir*

College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Saudi Arabia

Introduction: Generative AI tools can produce programming code that looks very similar to human-written code, which creates challenges in programming education. Students may use these tools inappropriately for their programming assignments, and there currently are not reliable methods to detect AI-generated code. It is important to address this issue to protect academic integrity while allowing the constructive use of AI tools. Previous studies have explored ways to detect AI-generated text, such as analyzing structural differences, embedding watermarks, examining specific features, or using fine-tuned language models. However, certain techniques, like prompt engineering, can make AI-generated code harder to identify.

Methods: To tackle this problem, this article suggests a new approach for instructors to handle programming assignment integrity. The idea is for instructors to use generative AI tools themselves to create example AI-generated submissions (pseudo-AI submissions) for each task. These pseudo-AI submissions, shared along with the task instructions, act as reference solutions for students. In the presence of pseudo-AI submissions, students are made aware that submissions resembling these examples are easily identifiable and will likely be flagged for lack of originality. On one side, this transparency removes the perceived advantage of using generative AI tools to complete assignments, as their output would closely match the provided examples, making it obvious to instructors. On the other side, the presence of these pseudo-AI submissions reinforces the expectation for students to produce unique and personalized work, motivating them to engage more deeply with the material and rely on their own problem-solving skills.

Results: A user study indicates that this method can detect AI-generated code with over 96% accuracy.

Discussion: The analysis of results shows that pseudo-AI submissions created using AI tools do not closely resemble student-written code, suggesting that the framework does not hinder students from writing their own unique solutions. Differences in areas such as expression assignments, use of language features, readability, efficiency, conciseness, and clean coding practices further distinguish pseudo-AI submissions from student work.

KEYWORDS
programming code plagiarism detection, detecting AI-generated code, generative AI tools, large language models (LLMs), programming code similarity

1 Introduction

With the rise of advanced large language models (LLMs), programmers increasingly use generative AI tools, such as ChatGPT, to assist with coding tasks (Bucaioni et al., 2024; Xu and Sheng, 2024). These tools, trained on extensive datasets, are highly proficient in understanding multiple programming languages. They excel at identifying bugs, offering solutions, and explaining complex programming concepts, which enhances productivity and facilitates learning (Ribeiro et al., 2023; Denny et al., 2022; Sarsa et al., 2022). However, these tools also present significant risks. A primary concern is their potential to generate code with security vulnerabilities (Tóth et al., 2024; Pearce et al., 2022).

Since the tools often prioritize syntax and functionality over secure practices, they may inadvertently produce code that exposes sensitive data or is susceptible to cyberattacks. Additionally, there are legal and ethical concerns regarding the use of these tools, particularly around potential copyright infringement.[1] Because generative AI tools learn from vast repositories of existing code, there is a risk that they might produce code that closely resembles proprietary or copyrighted material,[2] leading to possible legal disputes (Bucaioni et al., 2024; Xu and Sheng, 2024). To mitigate these risks, programmers and educators must implement safeguards, such as conducting thorough code reviews and tests to ensure the security and legality of AI-generated code. Promoting awareness of ethical coding practices and intellectual property laws among users can also help reduce the likelihood of these challenges.

In addition to this, another growing concern in the education sector is the use of generative AI tools by students to complete programming assignments and other homework tasks (Ghimire and Edwards, 2024; Becker et al., 2023; Denny et al., 2024). Although these tools can provide valuable assistance in generating code snippets, which can be especially helpful for beginners or students struggling with programming concepts (Biswas, 2023; Estévez-Ayres et al., 2024), they also pose significant risks. One key issue is that over-reliance on these tools can hinder the development of critical thinking and problem-solving skills, which are essential for tackling coding challenges independently (Jukiewicz, 2024). If students solve and submit programming tasks through generative AI tools without understanding the underlying concepts, it raises ethical concerns, as this could be considered academic dishonesty–students receiving credit for work they didn't genuinely complete. In response, some educational institutions have banned the use of generative AI tools for programming assignments. However, distinguishing between human-written and AI-generated code remains a challenge due to the lack of advanced detection frameworks. This differentiation is essential not only to ensure code quality, reliability, and security but also to address legal and ethical standards. Educators must find effective ways to identify instances of cheating and plagiarism enabled by these tools, as such detection is critical for upholding academic integrity.

Current AI text detection tools, such as GPTZero[3] and OpenAI Text Classifier,[4] face challenges in accurately distinguishing between AI-generated and human-written code (Xu et al., 2024). One reason for this limitation is that these tools are primarily trained on natural language text, not programming code. Traditional plagiarism detection tools like MOSS and JPlag (Orenstrakh et al., 2023), which compare submissions for direct copying, also struggle with the rise of generative AI. Modern AI tools can produce numerous unique solutions for the same problem in a non-deterministic manner, making it difficult for conventional methods to reliably identify cases of cheating. Recent advancements have explored the use of AI itself to detect AI-generated code, showing promise for improved accuracy (Nguyen et al., 2024; Xu and Sheng, 2024; Hoq et al., 2024). However, there remains a pressing need for new frameworks and tools specifically designed to identify AI-generated code to uphold fairness and integrity in academic assessments and software development practices.

Earlier research has largely focused on identifying structural differences between human-written and AI-generated code using supervised classification techniques (Nguyen et al., 2024; Xu and Sheng, 2024; Hoq et al., 2024). However, this process is complicated by students' ability to manipulate AI outputs through prompt engineering, disabling features that make the code classifiable (Hoq et al., 2024). To address these challenges, this paper introduces a novel framework for programming instructors aimed at preserving assignment integrity in the era of generative AI tools. The framework involves instructors using generative AI tools to produce a set of pseudo-AI submissions for each programming task (see Figure 1). These submissions, along with task descriptions, are shared with students as reference solutions. On one side, this transparency removes the perceived advantage of using generative AI tools to complete assignments, as their output would closely match the provided pseudo-AI submissions making it obvious to instructors. On the other side, the presence of these pseudo-AI submissions reinforces the expectation for students to produce unique and personalized work, motivating them to engage more deeply with the material and rely on their own problem-solving skills. Instructors can also integrate pseudo-AI submissions into plagiarism detection tools, allowing students to compare their work against these examples at submission. This approach encourages students to use the pseudo-AI submissions as a benchmark for originality rather than as a source for copying, fostering independent problem-solving skills. The framework is designed to shift student reliance from generative AI tools toward genuine engagement with the material and development of unique solutions. To evaluate the effectiveness of this framework, three key research questions need to explore:

- Research question 1: Do pseudo-AI submissions restrict the range of possible solutions, making it harder for students to produce distinct and non-similar work? We addressed this question in Section 3.1.
- Research question 2: given the non-deterministic nature of AI-generated code, how many distinct pseudo-AI submissions does an instructor need to provide in order to determine whether students have used generative AI tools to complete their work? We addressed this question in Section 3.2.
- Research question 3: What is the effectiveness of this framework for shorter or longer programming tasks? We address this question in Section 3.3.

2 Related word

Generative AI tools like ChatGPT and Claude,[5] while primarily designed for natural language tasks, have proven highly effective in

1 https://felixreda.eu/2021/07/github-copilot-is-not-infringing-your-copyright/

2 https://www.techtarget.com/searchsoftwarequality/news/252526359/Developers-warned-GitHub-Copilot-code-may-be-licensed

3 https://gptzero.me/

4 https://platform.openai.com/ai-text-classifier

5 https://claude.ai/

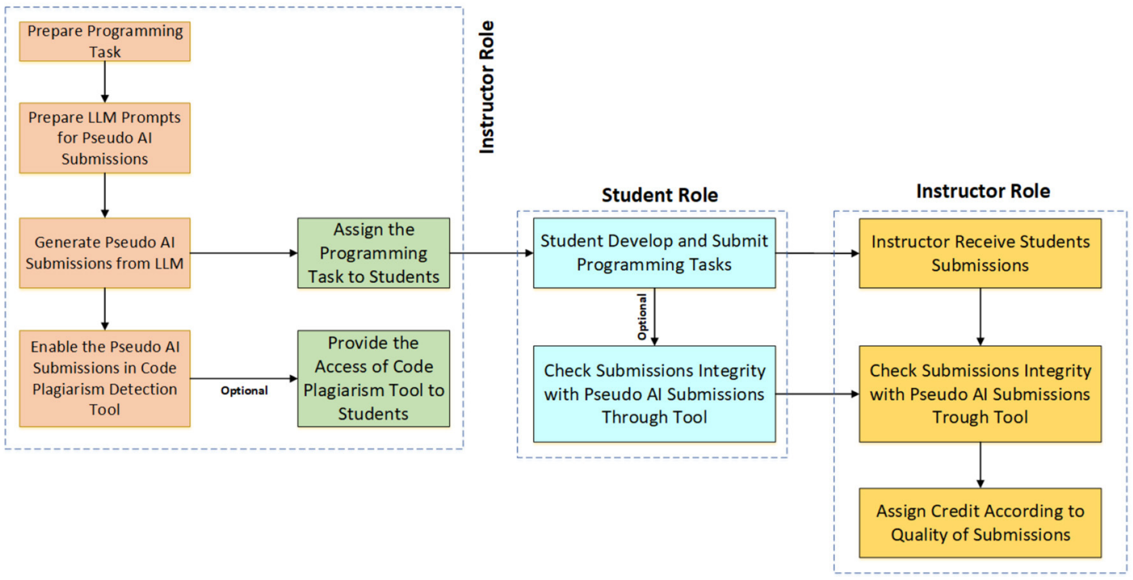

Instructor Role
Prepare Programming Task
Prepare LLM Prompts for Pseudo AI Submissions
Generate Pseudo AI Submissions from LLM
Enable the Pseudo AI Submissions in Code Plagiarism Detection Tool
Optional
Assign the Programming Task to Students
Provide the Access of Code Plagiarism Tool to Students
Student Role
Student Develop and Submit Programming Tasks
Optional
Check Submissions Integrity with Pseudo AI Submissions Through Tool
Instructor Role
Instructor Receive Students Submissions
Check Submissions Integrity with Pseudo AI Submissions Trough Tool
Assign Credit According to Quality of Submissions

FIGURE 1
The architecture of the pseudo-AI submission framework.

code-related applications. This effectiveness is due to their training on vast datasets of source code and technical documentation. These tools can handle tasks such as code completion, debugging, and synthesizing new code. Among these, OpenAI's Codex is particularly noteworthy. As a specialized extension of GPT-3 tailored for programming, Codex excels in code generation and understanding, making it a valuable resource for developers (Brown et al., 2020). A practical application of Codex is GitHub Copilot,[6] co-developed by GitHub, OpenAI, and Microsoft. Copilot provides real-time code suggestions based on the user's context, analyzing active files and related files to generate relevant solutions. Other tools like CodeGen (Nijkamp et al., 2022), Amazon CodeWhisperer,[7] and CodeGeeX (Zheng et al., 2023) also contribute to the field, showcasing the growing integration of AI in software development by converting natural language inputs into efficient code snippets. These advancements are revolutionizing how developers approach coding, making processes faster and more accessible.

Traditional methods in machine learning have played a significant role in distinguishing human-generated text from machine-generated content. Approaches such as bag-of-words combined with logistic regression have been effective, as demonstrated in Solaiman et al. (2019), where researchers differentiated GPT-2 outputs from human writing. Other methods, like log probability-based approaches, also show promise. For instance, TGM and GLTR (Chowdhery et al., 2023) leverage statistical techniques to identify patterns unique to AI-generated text. DetectGPT (Mitchell et al., 2023) further refines this process by combining log probabilities with random perturbations from pre-trained models like T5. Another widely used technique is fine-tuning pre-trained language models, exemplified by GROVER (Zellers et al., 2019) and RoBERTa (Uchendu et al., 2020), both of which achieve high accuracy in detecting AI-generated content. For example, a fine-tuned RoBERTa model has demonstrated ∼95% accuracy in distinguishing GPT-2-generated web pages. Additionally, researchers have introduced innovative methods like watermarking frameworks for large language models (LLMs), which embed detectable signals into generated content while keeping them imperceptible to human readers (Kirchenbauer et al., 2023).

In the realm of code plagiarism detection, tools like MOSS and JPlag remain popular among educators (Prechelt et al., 2002). These tools compare token similarities between programs to identify cases of copying. For example, the CloSpan data mining algorithm, proposed in Kechao et al. (2012), enhances plagiarism detection by analyzing similar code segments and visualizing results, offering greater precision than traditional tools like MOSS. Another advanced tool uses the XGBoost incremental learning algorithm, which reaches high accuracy when detecting plagiarism in academic and industry scenarios (Huang et al., 2019). However, these traditional tools face limitations in detecting code generated by generative AI tools. Since AI-generated code is produced through stochastic processes, it lacks the deterministic structure that traditional models rely on for similarity detection. Despite recent advances in identifying AI-generated text (Nguyen et al., 2024; Xu and Sheng, 2024; Hoq et al., 2024), there remains a gap in effectively distinguishing between human-written and AI-generated code. Addressing this challenge is critical as the use of generative AI tools becomes increasingly prevalent in programming and education.

3 Pseudo AI submission framework

In our proposed framework, instructors use generative AI tool to create a set of pseudo-AI solutions for each programming task

6 https://github.com/features/copilot
7 https://aws.amazon.com/q/developer/

assigned to students. These pseudo-AI submissions, along with the task descriptions, are then provided to the students as reference solutions (see Figure 1). By making these pseudo-AI submissions available, the instructors discourage students from using generate AI tools to generate their programming solutions directly. If they do, their submissions will have a high similarity to the pseudo-AI solutions, making detection straightforward. The aim is to prevent students from relying on generative AI tools for their work and instead encourage them to engage with the material and develop their own code.

We use student-written code from a publicly available dataset (Ljubovic, 2020) to analyze the effectiveness of the proposed framework. This dataset contains code from an introductory programming course in the C language, covering 48 different programming problems. For our experiment, we selected 28 problems from this collection: 22 are long problems requiring more than 15 lines of code, and six are short problems with less than 16 lines of code. These programming problems cover fundamental C language concepts, such as functions, variable declarations, data types, conditional statements, string (char array), arrays, and iterations. To analyze the effectiveness of our framework, we developed a dataset containing the following three sets:

- Descriptions of programming tasks provided by instructors to their students.
- Original student submissions for these programming tasks. To ensure a fair analysis, we selected student submissions from the years 2016 and 2017, before the availability of popular generate AI tools.
- Pseudo-AI submissions for each programming task generated by a generative AI tool (ChatGPT[8]) using different prompts.

To generate the pseudo-AI submissions, we used the ChatGPT GUI, as students of introductory programming are more likely to use the online ChatGPT interface rather than the GPT API. A basic ChatGPT prompt consists of: "Generate C/Java code using the following problem description: [problem statement]. The function prototype is given: [function prototype]." We included the function prototype if it was provided along with the problem statement to ensure ChatGPT had the same information as the students when constructing the solutions. Table 1 lists all prompts we used for generating pseudo-AI submissions.

To generate each instance of a solution for a specific task, the last two prompts of Table 1 regenerate the solutions using the code from the first prompt with the following instruction: "Regenerate C/Java code using the given [C/Java code]. The regenerated code should not be similar to the given C/Java code." We used prompt 13 to regenerate code multiple times. However, we observed that only the first regeneration produced code different from the given code. Subsequent regenerations did not result in significantly different code. We collected ChatGPT-generated solutions for each task, which are used for comparison with the student written submissions. Our objective is solely to discourage students from directly generating solutions using generative AI tools, so we did not verify whether the ChatGPT produced correct code for the tasks.

8 https://chatgpt.com/

TABLE 1 Prompts used for generating the pseudo-AI submissions from generative AI tool.

Prompt ID	Prompt statement
Prompt 1	Generate C/Java code using the following problem description [problem description] and function prototype [function prototype]
Prompt 2	Prompt 1 + The code should look like how early programming students write code
Prompt 3	Prompt 1 + Write the code as a novice programmer
Prompt 4	Prompt 1 + Avoid complications while writing the code
Prompt 5	Prompt 1 + Do not compact the code. Generate a long code
Prompt 6	Prompt 1 + Write it as an introductory programming student would
Prompt 7	Prompt 1 + Do not compact the code. The code should look like how early programming students write code
Prompt 8	Prompt 1 + Create many functions. Create an individual function for each subtask
Prompt 9	Prompt 1 + Create a separate function for each mathematical statement used in the code
Prompt 10	Prompt 1 + Write the conditional statements using only the switch statement. Do not use if statements
Prompt 11	Prompt 1 + Write the conditional statements using the ternary operator. Do not use if statements
Prompt 12	Prompt 1 + Generate the code without creating any function or procedure
Prompt 13	Regenerate C/Java code using the given [C/Java code]. The regenerated code should not be similar to the given C/Java code
Prompt 14	Regenerate again C code using the given [C/Java code] of Prompt 13. The regenerated code should not be similar to the given C/Java code

3.1 Code similarity between student-written codes and AI-generated codes (research question 1)

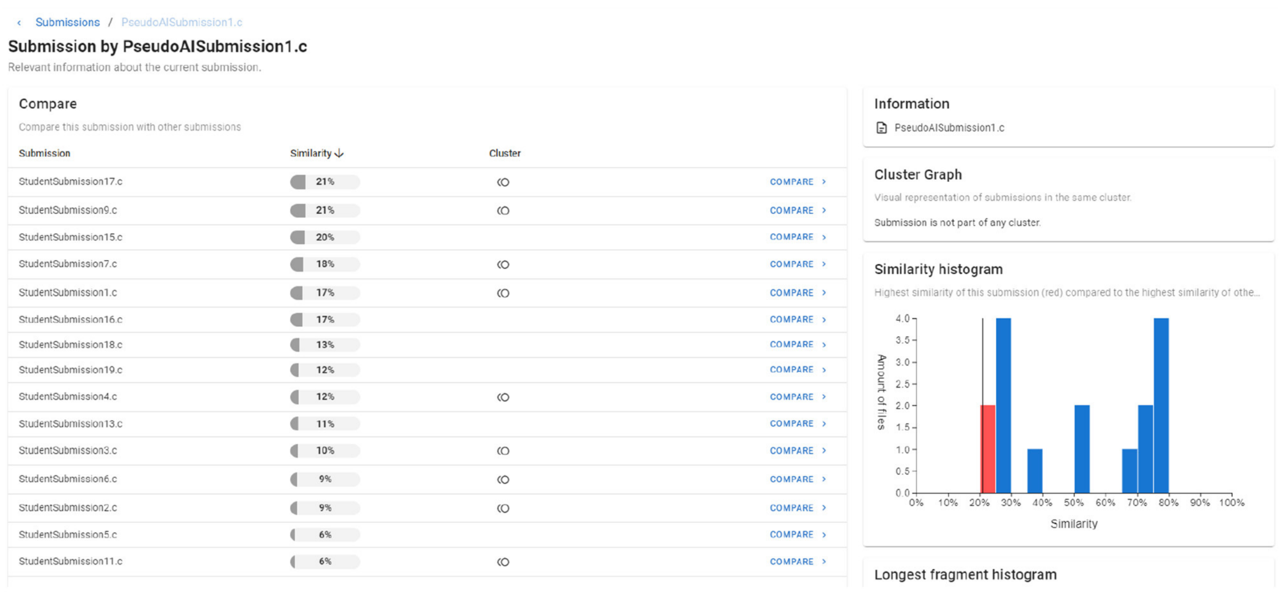
An important research question we need to address is whether the presence of pseudo-AI submissions impact the range of student solutions, making it harder for students to come up with unique and different solutions. To address this research question, we compared the code similarity between the pseudo-AI submissions and actual student-written submissions using Dolos (Maertens et al., 2024, 2022). Dolos converts the codes into abstract syntax trees (ASTs) and calculates similarity based on the coverage of unique AST fingerprints. High similarity scores indicate that the pseudo-AI submissions are too similar to student-written codes, potentially restricting the students' creative space. Tables 2, 3 show the results. The results indicate that pseudo-AI submissions do not have high similarity with student-written codes. Figure 2 shows a sample code similarity using Dolos of a pseudo-AI submission with a student-written code. To further understand why the pseudo-AI submissions, have low similarity with student-written codes, we analyzed their codes and observed several key structural differences between two types:

TABLE 2 Code similarity between student-written codes and pseudo-AI submissions.

Comparison type	Avg. of minimum similarity	Avg. of median similarity	Avg. of maximum similarity
Student code -to- student code	23%	39%	100%
Student code -to- pseudo code	3%	11%	22%
Pseudo code -to- pseudo code	38%	65%	86%

TABLE 3 Code similarity of each (AI generative tool) prompt with the student-written submissions.

Prompt	Avg. of minimum similarity	Avg. of median similarity	Avg. of maximum similarity
Prompt 1	2%	10%	20%
Prompt 2	2%	16%	27%
Prompt 3	3%	17%	28%
Prompt 4	2%	13%	25%
Prompt 5	2%	12%	24%
Prompt 6	2%	12%	23%
Prompt 7	2%	11%	21%
Prompt 8	2%	9%	16%
Prompt 9	2%	8%	14%
Prompt 10	2%	8%	14%
Prompt 11	2%	9%	16%
Prompt 12	2%	9%	18%
Prompt 13	2%	10%	19%
Prompt 14	2%	10%	20%



‹ Submissions / PseudoAISubmission1.c

Submission by PseudoAISubmission1.c

Relevant information about the current submission.

Compare

Compare this submission with other submissions

Submission	Similarity ↓	Cluster	
StudentSubmission17.c	21%	⦾	COMPARE ›
StudentSubmission9.c	21%	⦾	COMPARE ›
StudentSubmission15.c	20%		COMPARE ›
StudentSubmission7.c	18%	⦾	COMPARE ›
StudentSubmission1.c	17%	⦾	COMPARE ›
StudentSubmission16.c	17%		COMPARE ›
StudentSubmission18.c	13%		COMPARE ›
StudentSubmission19.c	12%		COMPARE ›
StudentSubmission4.c	12%	⦾	COMPARE ›
StudentSubmission13.c	11%		COMPARE ›
StudentSubmission3.c	10%	⦾	COMPARE ›
StudentSubmission6.c	9%	⦾	COMPARE ›
StudentSubmission2.c	9%	⦾	COMPARE ›
StudentSubmission5.c	6%		COMPARE ›
StudentSubmission11.c	6%	⦾	COMPARE ›

Information

PseudoAISubmission1.c

Cluster Graph

Visual representation of submissions in the same cluster.

Submission is not part of any cluster.

Similarity histogram

Highest similarity of this submission (red) compared to the highest similarity of othe...

Longest fragment histogram

FIGURE 2
The figure shows a sample code similarity using Dolos of a pseudo-AI submission with the student-written codes.

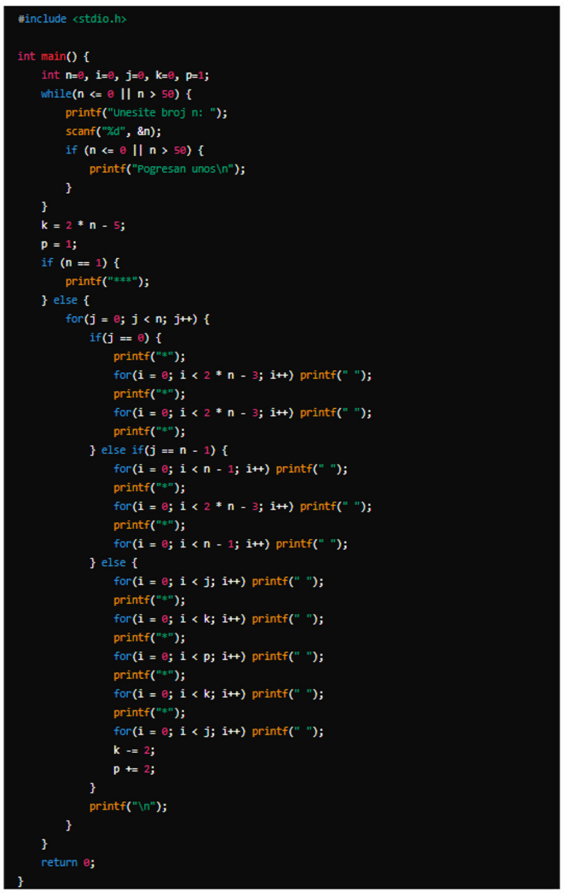
- Compact code: codes generated from generated AI tool are typically more compact and concise, effectively utilizing fewer lines and minimizing redundant code structures. In contrast, student-written codes often exhibit less compactness, with verbose implementations that include unnecessary lines and repetitive constructs. Figure 3 shows an example of student-written code and pseudo-AI submission using prompt 1 for a pattern drawing task. The pseudo-AI submission generated from a generated AI tool is more compact than the student-written code.

- Expression assignment: pseudo-AI submissions often assign complex expressions to variables before using them as conditions in later if-statements, whereas students tend to use the expressions directly, often duplicating them across multiple if-statements. Additionally, pseudo-AI submissions use a direct return statement following an if-return statement, omitting the need for an else-return statement that is common in student codes.
- Language features: pseudo-AI submissions are more likely to utilize newer language features. For instance, in Java, it favors the "foreach" loop over the traditional "for" loop, and in C++, it frequently uses the "auto" keyword for type inference.
- Readability: pseudo-AI submissions emphasize readability by employing meaningful words for identifiers, whereas student-written codes may use abbreviations or letters with ambiguous meanings.
- Efficiency: unlike student-written codes, which often assign a value to a Boolean variable before returning it, pseudo-AI submissions directly return the expression, eliminating the need for an additional assignment statement and enhancing code efficiency.
- Conciseness: Pseudo-AI submissions frequently use ternary operators as an alternative to multiple if-else conditions, allowing for more concise and streamlined code.
- Clean code: student-written codes often exhibit remnants of development iterations, such as commented out code snippets, which are absent in the pseudo-AI submissions. Additionally, pseudo-AI submissions do not include extraneous code, while student-written codes sometimes contains unreferenced variables or methods.

In short, pseudo-AI submissions simplify control flow and reduce overall code length. In our experiments, the similarity analysis between pseudo-AI submissions and student-written codes for a set of programming tasks demonstrates that codes generated from generative AI tools have a different structure than code written by early-year students. Thus, the proposed framework does not limit the solution space or create additional challenges for the students.

3.2 Number of pseudo AI submissions (research question 2)

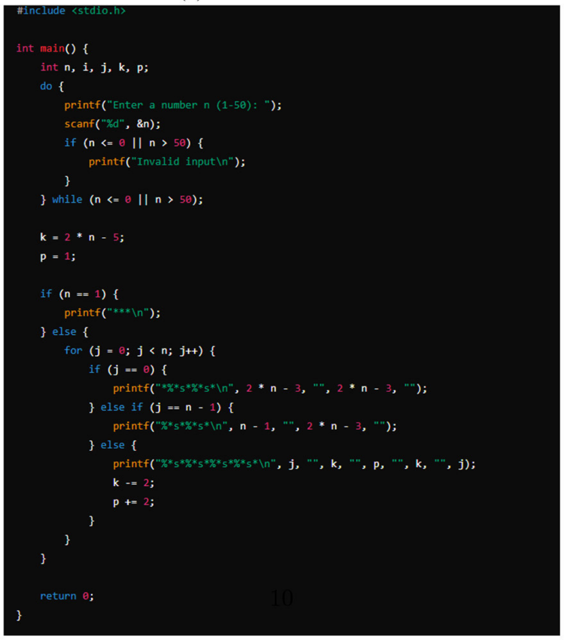
The analysis of our initial question reveals that codes generated from generative AI tools are significantly different from student-written codes. This finding leads us to our next question: How feasible is it for students to complicate the detection process by prompting generative AI tools to generate solutions distinct from pseudo-AI submissions? Specifically, we aim to determine the number of pseudo-AI submissions an instructor would need to generate to encompass the range of possible AI-generated code variations.



```c
#include <stdio.h>

int main() {
    int n=0, i=0, j=0, k=0, p=1;
    while(n <= 0 || n > 50) {
        printf("Unesite broj n: ");
        scanf("%d", &n);
        if (n <= 0 || n > 50) {
            printf("Pogresan unos\n");
        }
    }
    k = 2 * n - 5;
    p = 1;
    if (n == 1) {
        printf("***");
    } else {
        for(j = 0; j < n; j++) {
            if(j == 0) {
                printf("*");
                for(i = 0; i < 2 * n - 3; i++) printf(" ");
                printf("*");
                for(i = 0; i < 2 * n - 3; i++) printf(" ");
                printf("*");
            } else if(j == n - 1) {
                for(i = 0; i < n - 1; i++) printf(" ");
                printf("*");
                for(i = 0; i < 2 * n - 3; i++) printf(" ");
                printf("*");
                for(i = 0; i < n - 1; i++) printf(" ");
            } else {
                for(i = 0; i < j; i++) printf(" ");
                printf("*");
                for(i = 0; i < k; i++) printf(" ");
                printf("*");
                for(i = 0; i < p; i++) printf(" ");
                printf("*");
                for(i = 0; i < k; i++) printf(" ");
                printf("*");
                for(i = 0; i < j; i++) printf(" ");
                k -= 2;
                p += 2;
            }
            printf("\n");
        }
    }
    return 0;
}
```

(a)



```c
#include <stdio.h>

int main() {
    int n, i, j, k, p;
    do {
        printf("Enter a number n (1-50): ");
        scanf("%d", &n);
        if (n <= 0 || n > 50) {
            printf("Invalid input\n");
        }
    } while (n <= 0 || n > 50);

    k = 2 * n - 5;
    p = 1;

    if (n == 1) {
        printf("***\n");
    } else {
        for (j = 0; j < n; j++) {
            if (j == 0) {
                printf("*%*s*%*s*\n", 2 * n - 3, "", 2 * n - 3, "");
            } else if (j == n - 1) {
                printf("%*s*%*s*\n", n - 1, "", 2 * n - 3, "");
            } else {
                printf("%*s*%*s*%*s*%*s*\n", j, "", k, "", p, "", k, "", j);
                k -= 2;
                p += 2;
            }
        }
    }

    return 0;
}
```

(b)

FIGURE 3
The figures show an example of student-written code and pseudo-AI submission using prompt 1 for the pattern drawing task. The pseudo-AI submission generated from a generative AI tool is more compact than the student-written code. **(a)** Student-written code. **(b)** Pseudo-AI submission using Prompt 1.

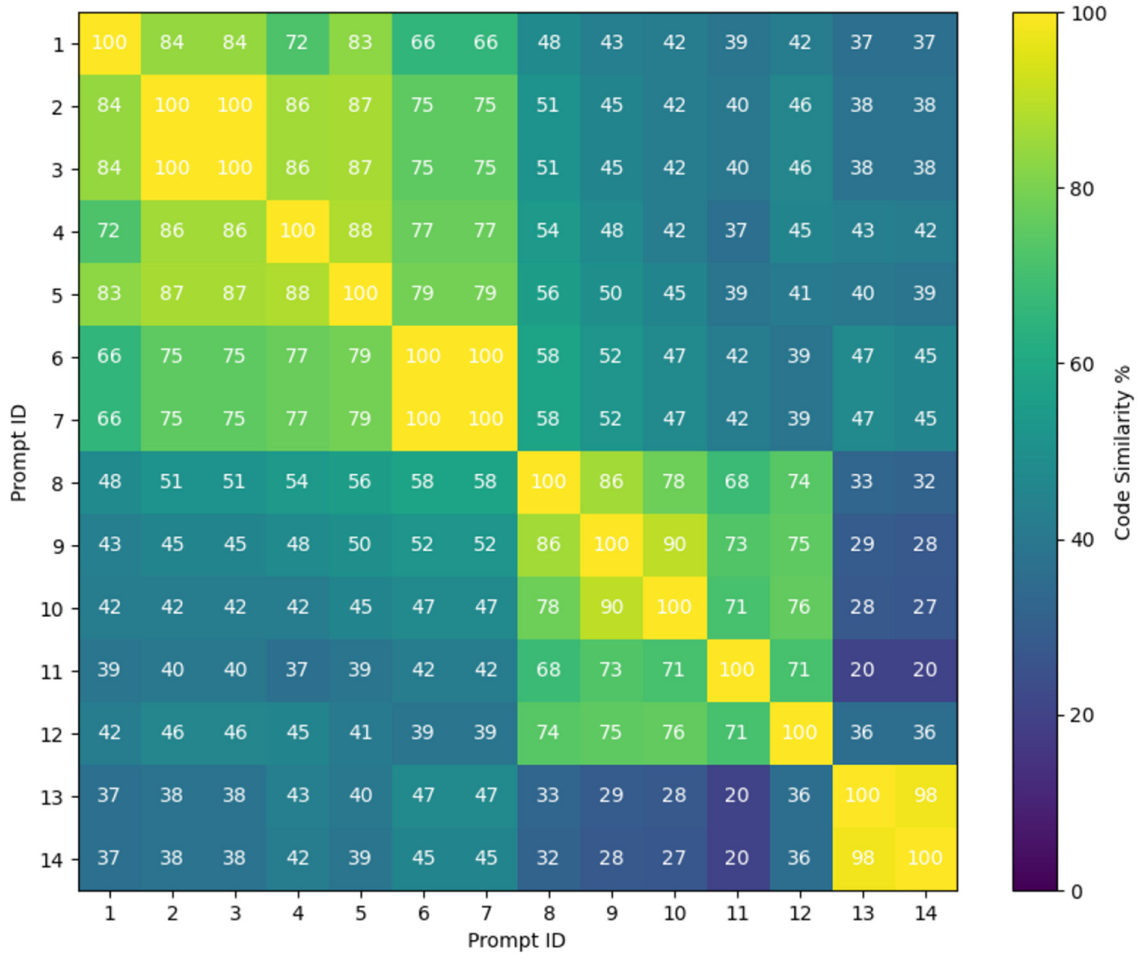


FIGURE 4
Pairwise prompts similarity using Heat Matrix.

To study this, we analyzed the similarity between student-written code and pseudo-AI submissions, which were created using prompt engineering with the prompts shown in Table 2. Table 2 presents the maximum, median, and minimum similarity scores between the pseudo-AI submissions for various programming tasks. High similarity between the pseudo-AI submissions indicates that, even if students attempt to complicate the detection process through prompt engineering to generate codes different from pseudo-AI submissions, the resulting codes would still be similar to the pseudo-AI submissions.

Next, we want to determine how many pseudo-AI submissions an instructor needs to maintain to discourage students from utilizing generative AI tools for generating programming solutions. According to the results shown in Figure 4, prompts 1 to 5 exhibit a strong code similarity of 83% among them. Therefore, it is unnecessary to keep all the AI-generated codes from these prompts; the instructor can retain just one. Similarly, prompts 6 and 7, as well as prompts 8, 9, 10, and 12, show strong code similarities within their respective clusters. Prompt 11 stands alone because it asks the generative AI tool to generate code using the ternary operator, which involves more complex logic compared to conditional statements like if, if-else, and switch. Figure 4 shows strong similarities between prompt 13 and prompt 14 because both prompts instruct the generative AI tool to regenerate the code using the code from prompt 1. While prompt 13 generates a code different from prompt 1, the regenerated code from prompt 14 has high similarity with prompt 13. This indicates that further regenerating the code does not provide any benefit.

Table 4 lists all the clusters among prompts. From the results, we can infer that even if an instructor retains a single prompt from each of the five clusters, they can still effectively prevent students from relying on generative AI tool to generate their programming solutions. In another experiment, we created a prompt designed to test whether a student can complicate the pseudo-AI submissions by asking the generative AI tools to generate code with extra features not requested in the problem description. The goal of this prompt was to produce longer and more complex code by including additional features. Examples of extra features requested from the generative AI tool include:

TABLE 4 (Generative AI tool) prompts clusters.

Cluster #	Prompts in the cluster	Minimum similarity between prompts
1	Prompt 1, Prompt 2, Prompt 3, Prompt 4, Prompt 5	72%
2	Prompt 6, Prompt 7	100%
3	Prompt 8, Prompt 9, Prompt 10, Prompt 12	74%
4	Prompt 11	100%
5	Prompt 13, Prompt 14	98%

The third column shows minimum similarity between prompts of the cluster.

TABLE 5 Effectiveness of the proposed framework on short and long programming tasks.

	Similarity	
Code length	Minimum	Maximum
Short code [1−15]	12%	48%
Long code [>15]	2%	18%

- Sorting arrays after every insertion or access to the array.
- Creating an option menu for starting, quitting, and handling each task described in the problem.
- Adding code to search for a specific number in each array after every access or insertion.
- I/O file handling code, if the instructor did not specify it in the programming task description, such as saving data to a file or reading data from a file.

The prompt used for this experiment was: "Generate the C/Java code based on the following problem description [problem description] and function prototype [function prototype]. Include the following extra features in the generated code [list of extra features]." We tested this prompt on all programming tasks and found that the codes generated by this prompt show an average maximum similarity of 42% with the 14 prompts listed in Table 1. This low similarity occurred because the resulting AI-generated code contains extra features that are not present in the other pseudo-AI submissions.

To address this issue, instructors can review the code and remove any unnecessary functionality not specified in the problem description. To test this, we developed another prompt for the instructor with the following instruction: "Given the following [C/Java code], remove any extra features or functions that are not present in the problem description [problem description]. Do not modify the given code; simply remove the additional features." We applied this prompt to AI-generated code with extra features and compared the similarity of the generated code with the pseudo-AI submissions. The results showed an average maximum similarity of 90% with prompt 1. This indicates that even if students complicate the AI-generated code by adding extra features, the instructor can still detect AI-generated code by reviewing the codes of the tasks.

TABLE 6 Effectiveness of the proposed framework on user study.

	Similarity		
Submission type	Minimum	Median	Maximum
Assignment 1 + students implementations	3%	14%	25%
Assignment 1 + AI-generated codes	21%	59%	87%
Assignment 2 + students implementations	3%	13%	24%
Assignment 2 + AI-generated codes	20%	60%	88%

3.3 Effectiveness of proposed framework on short and long programming tasks (research question 3)

For this research question, we aim to evaluate the effectiveness of our framework with respect to the length of the programming task. Table 5 presents the results with varying code lengths. According to these results, smaller programming tasks, those with less than 16 lines of code, exhibit significantly higher similarity between student-written codes and pseudo-AI submissions. This is because the solution space for these smaller problems is more limited, leading to fewer possibilities for variation and consequently higher similarity with the AI-generated codes. Thus, detecting differences in these cases becomes more challenging, particularly when the code size is very small, as AI-generated code may share similar code structures. Despite this, novice programming students still exhibit distinct code structures that experts and educators can easily recognize. These structures include placing values before variable names in expressions, using multiple if conditions instead of else-if conditions, and other novice patterns mentioned earlier in Section 3.1.

However, the scenario changes for longer programming tasks with more than 15 lines of code, which show low similarity between student-written codes and pseudo-AI submissions. This is because, for longer programming codes, student-written codes often exhibit unoptimized programming practices. In contrast, pseudo-AI submissions are more optimized and compact (see Figure 3). The generative AI tools tend to use more efficient return statements, avoid unnecessary else statements, and eliminate unreachable else statements in conditional structures, resulting in a more professional and expert-like programming style.

4 User study

To analyze the effectiveness of the proposed framework, we conducted a user study in the "Introduction to Computer Programming" course with 43 students across two sections. The participants in our study were students who had not yet taken any data structures or algorithms courses. However, they had completed an "Introduction to Computer Science" course, which covered foundational topics such as software and hardware concepts, input/output devices, HTML, cybersecurity, artificial intelligence, databases, and operating systems. The students were familiar with using generative AI tools to generate programming assignment solutions. Specifically, they knew how to construct prompts to direct AI tools in generating solutions but lacked advanced knowledge of programming concepts beyond introductory topics. The study focused on two assignments. Both assignments were lengthy, requiring more than 16 lines of code.

The programming tasks assigned to students were designed to cover fundamental programming concepts typically introduced in an introductory programming course. Each task consisted of a problem statement requiring students to implement a solution in the C programming language. These tasks varied in complexity, ranging from simple computational problems to more structured problems requiring modularization and efficient code organization. For example, some tasks involved basic arithmetic calculations, string manipulations, and data processing, while others required implementing iterative and conditional logic to solve pattern generation or sorting problems. The primary goal of these tasks was to assess students' understanding of fundamental programming constructs and their ability to write logical, structured, and efficient code. The programming concepts involved in these tasks included variable declarations, data types, input and output handling, conditional statements (if-else, switch), loops (for, while, do-while), functions, arrays, and string manipulations. Some tasks required students to break down problems into multiple functions, reinforcing modularity and code reusability. Additionally, certain problems encouraged students to implement simple algorithmic logic, such as searching for elements in an array or processing string inputs according to specified rules.

Although the students at this level do not have data structure or algorithm skills, they were still advised to submit the assignment in C language without using any advanced data structures such as linked lists, hashing, binary search trees, etc. Students were asked to submit two versions of their code for each assignment: one based on their own work and the other generated through a generative AI tool. Additionally, they were required to submit the AI-generated code along with the prompt they used to generate it. For a fair analysis, students were informed that their submissions would be compared with the AI-generated codes (pseduo AI-submissions) through code similarity. The instructor advised the students to use the generative AI tool for one of their submissions and assured them that it would not be considered academic dishonesty if their AI-generated submissions matched with the pseudo-AI submissions. Furthermore, to make pseudo-AI submissions detection more challenging, the instructors informed students that they would receive bonus marks if their AI-generated submissions had low similarity to the pseudo-AI submissions.

The students submitted their assignments on the due dates. For the submissions where students used their own implementations, we found low similarity between their work and the pseudo-AI submissions. However, for the submission that were generated from generative AI tool, the code similarity detection tool showed high similarities between the student submissions and the pseudo-AI submissions. Table 6 presents the results of both submissions. Table 7 shows a list of the top five prompts from the students where we found low similarity between the pseudo-AI submission and the students' codes generated from generative AI tool. The lowest similarity was found with the prompt "Generate a correct C program given the following assignment description. Declare and use only two-dimensional arrays while implementing the code." In this prompt the students asked generative AI tool to implement the code using only two-dimensional arrays. Nevertheless, this code still had high similarity with prompt 1, at 82%. In the majority of prompts where we found high similarity with pseudo-AI submissions, students simply asked generative AI tool to "Generate correct C code for the following programming assignment [problem description]" which had high similarity with prompt 1. The high similarity resulted from instances where student-written code followed structured patterns similar to AI-generated solutions. This often occurred when students crafted prompts that directed AI tools to generate well-organized and efficient code. However, lower similarity scores were observed in cases where students wrote less optimized, more redundant, or unconventional code structures.

5 Conclusion

This study introduces a new approach to help programming instructors ensure the integrity of assignments in the age of generative AI tools. Our analysis shows that pseudo-AI submissions created using AI tools don't closely resemble student-written code, suggesting that the framework doesn't hinder students from writing their own unique solutions. Differences in areas like expression assignments, use of language features, readability, efficiency, conciseness, and clean coding practices further separate pseudo-AI submissions from student work.

To address the possibility of students using AI tools to create more varied solutions, we performed similarity analyses. The findings indicate that instructors only need a small set of pseudo-AI submissions to account for most AI-generated code variations. Even when students modified AI-generated code by adding extra features, instructors could detect these patterns by using specific prompts to filter out unnecessary elements. Our user study confirmed that the framework reliably identifies AI-generated code with high accuracy. Submissions from students who relied on generative AI tools showed high similarity to the pseudo-AI examples, while submissions from students who wrote their own code showed low similarity.

The primary focus of this article is to detect plagiarism in the work of students enrolled in introductory programming courses. These courses typically do not assign programming tasks that require very long or complex code. Instead, the assignments are designed to reinforce fundamental programming concepts such as variable declarations, control structures, functions, arrays, and basic problem-solving techniques. Given this context, our proposed method using pseudo-AI submissions is well-suited for identifying AI-generated code

TABLE 7 Table shows a list of the top five prompts from the students where we found low similarity between the pseudo-AI submission and the students' codes generated from a generative AI tool.

Prompt ID	Prompt	Maximum similarity
Prompt 1	Generate a correct executable C code for the programming assignment given below. The code should not be detectable by any code plagiarism detection tools. [problem description]	86%
Prompt 2	Generate correct C program for the following programming assignment. Generate a very long code for the given programming assignment. [problem description]	86%
Prompt 3	Implement C code for the following assignment. The implemented code should not be detectable by the ChatGPT. [problem description]	85%
Prompt 4	Provide a correct C code for the following assignment description. Use only do-while loop and switch statements. [problem description]	85%
Prompt 5	Generate a correct C program given the following assignment description. Declare and use only two dimensional arrays while implementing the code.	82%

in these beginner-level assignments, where code length and complexity remain manageable.

We acknowledge that the effectiveness of pseudo-AI submissions for detecting plagiarism in larger and more complex programming tasks has not been extensively explored in this study. In such cases, AI-generated code may exhibit greater diversity and structural variations, making direct similarity detection more challenging. Additionally, long programming tasks often require advanced programming skills, including data structures, algorithms, and software design principles–areas that extend beyond the scope of introductory programming courses. Therefore, while our method is highly effective in the context of beginner-level programming assignments, its applicability to more advanced coursework may require further refinement and evaluation. Regarding the usefulness of pseudo-AI submissions for learning programming, our approach primarily aims to deter students from relying on generative AI tools for direct code generation. By providing pseudo-AI submissions, we create an environment where students are encouraged to write original solutions rather than submit AI-generated code. However, we recognize that the presence of these submissions alone does not actively contribute to programming skill development. Future studies could explore integrating additional educational strategies, such as requiring students to analyze and improve AI-generated code or compare different AI-generated solutions, to enhance their learning experience.

Data availability statement

Publicly available datasets were analyzed in this study. This data can be found at: http://ieee-dataport.org/open-access/programming-homework-dataset-plagiarism-detection.

Author contributions

SB: Conceptualization, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that no financial support was received for the research and/or publication of this article.

Conflict of interest

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Publisher's note

References

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., Santos, E. A., et al. (2023). "Programming is hard-or at least it used to be: educational opportunities and challenges of AI code generation," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (New York, NY: ACM), 500–506. doi: 10.1145/3545945.3569759

Biswas, S. (2023). Role of chatgpt in computer programming. *Mesopotam. J. Comput. Sci.* 2023, 9–15. doi: 10.58496/MJCSC/2023/002

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., et al. (2020). Language models are few-shot learners. *Adv. Neural Inf. Process Syst.* 33, 1877–1901. doi: 10.5555/3495724.3495883

Bucaioni, A., Ekedahl, H., Helander, V., and Nguyen, P. T. (2024). Programming with chatgpt: how far can we go? *Mach. Learn. Appl.* 15:100526. doi: 10.1016/j.mlwa.2024.100526

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., et al. (2023). Palm: scaling language modeling with pathways. *J. Mach. Learn. Res.* 24, 1–113. doi: 10.5555/3648699.3648939

Denny, P., Prather, J., Becker, B. A., Finnie-Ansley, J., Hellas, A., Leinonen, J., et al. (2024). Computing education in the era of generative AI. *Commun. ACM* 67, 56–67. doi: 10.1145/3624720

Denny, P., Sarsa, S., Hellas, A., and Leinonen, J. (2022). Robosourcing educational resources-leveraging large language models for learnersourcing. *arXiv* [Preprint]. arXiv:2211.04715. doi: 10.48550/arXiv.2211.04715

Estévez-Ayres, I., Callejo, P., Hombrados-Herrera, M. Á., Alario-Hoyos, C., and Delgado Kloos, C. (2024). Evaluation of LLM tools for feedback generation in a course on concurrent programming. *Int. J. Arti. Intell. Educ.* 1–17. doi: 10.1007/s40593-024-00406-0

Ghimire, A., and Edwards, J. (2024). "Coding with AI: how are tools like chatgpt being used by students in foundational programming courses," in *International Conference on Artificial Intelligence in Education* (Cham: Springer), 259–267. doi: 10.1007/978-3-031-64299-9_20

Hoq, M., Shi, Y., Leinonen, J., Babalola, D., Lynch, C., Price, T., et al. (2024). "Detecting chatgpt-generated code submissions in a cs1 course using machine learning models," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (New York, NY: ACM), 526–532. doi: 10.1145/3626252.3630826

Huang, Q., Fang, G., and Jiang, K. (2019). "An approach of suspected code plagiarism detection based on xgboost incremental learning," in *2019 International Conference on Computer, Network, Communication and Information Systems (CNCI 2019)* (Dordrecht: Atlantis Press), 269–276. doi: 10.2991/cnci-19.2019.40

Jukiewicz, M. (2024). The future of grading programming assignments in education: the role of chatgpt in automating the assessment and feedback process. *Think. Skills Creat.* 52:101522. doi: 10.1016/j.tsc.2024.101522

Kechao, W., Tiantian, W., Mingkui, Z., Zhifei, W., and Xiangmin, R. (2012). "Detection of plagiarism in students' programs using a data mining algorithm," in *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology* (Changchun: IEEE), 1318–1321. doi: 10.1109/ICCSNT.2012.6526164

Kirchenbauer, J., Geiping, J., Wen, Y., Katz, J., Miers, I., Goldstein, T., et al. (2023). "A watermark for large language models," in *International Conference on Machine Learning* (Cambridge, MA: PMLR), 17061–17084.

Ljubovic, V. (2020). *Programming Homework Dataset for Plagiarism Detection*. Piscataway, NJ: IEEE Dataport.

Maertens, R., Van Neyghem, M., Geldhof, M., Van Petegem, C., Strijbol, N., Dawyndt, P., et al. (2024). Discovering and exploring cases of educational source code plagiarism with dolos. *SoftwareX* 26:101755. doi: 10.1016/j.softx.2024.101755

Maertens, R., Van Petegem, C., Strijbol, N., Baeyens, T., Jacobs, A. C., Dawyndt, P., et al. (2022). Dolos: language-agnostic plagiarism detection in source code. *J. Comput. Assist. Learn.* 38, 1046–1061. doi: 10.1111/jcal.12662

Mitchell, E., Lee, Y., Khazatsky, A., Manning, C. D., and Finn, C. (2023). "Detectgpt: zero-shot machine-generated text detection using probability curvature," in *International Conference on Machine Learning* (Cambridge, MA: PMLR), 24950–24962.

Nguyen, P. T., Di Rocco, J., Di Sipio, C., Rubei, R., Di Ruscio, D., and Di Penta, M. (2024). Gptsniffer: a codebert-based classifier to detect source code written by chatgpt. *J. Syst. Softw.* 214:112059. doi: 10.1016/j.jss.2024.112059

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., et al. (2022). Codegen: an open large language model for code with multi-turn program synthesis. *arXiv* [Preprint]. arXiv:2203.13474. doi: 10.48550/arXiv.2203.13474

Orenstrakh, M. S., Karnalim, O., Suarez, C. A., and Liut, M. (2023). Detecting LLM-generated text in computing education: a comparative study for chatgpt cases. *arXiv* [Preprint]. arXiv:2307.07411. doi: 10.48550/arXiv.2307.07411

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). "Asleep at the keyboard? Assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)* (San Francisco, CA: IEEE), 754–768. doi: 10.1109/SP46214.2022.9833571

Prechelt, L., Malpohl, G., and Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPLAG. *J. Univers. Comput. Sci.* 8:1016. doi: 10.3217/jucs-008-11-1016

Ribeiro, F., de Macedo, J. N. C., Tsushima, K., Abreu, R., and Saraiva, J. (2023). "GPT-3-powered type error debugging: investigating the use of large language models for code repair," in *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY: ACM), 111–124. doi: 10.1145/3623476.3623522

Sarsa, S., Denny, P., Hellas, A., and Leinonen, J. (2022). "Automatic generation of programming exercises and code explanations using large language models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research, Vol. 1* (New York, NY: ACM), 27–43. doi: 10.1145/3501385.3543957

Solaiman, I., Brundage, M., Clark, J., Askell, A., Herbert-Voss, A., Wu, J., et al. (2019). Release strategies and the social impacts of language models. *arXiv* [Preprint]. arXiv:1908.09203.doi: 10.48550/arXiv.1908.09203

Tóth, R., Bisztray, T., and Erdodi, L. (2024). LLMs in web-development: evaluating LLM-generated PHP code unveiling vulnerabilities and limitations. *arXiv* [Preprint]. arXiv:2404.14459. doi: 10.48550/arXiv.2404.14459

Uchendu, A., Le, T., Shu, K., and Lee, D. (2020). "Authorship attribution for neural text generation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Stroudsburg, PA: Association for Computational Linguistics (ACL)), 8384–8395. doi: 10.18653/v1/2020.emnlp-main.673

Xu, Z., and Sheng, V. S. (2024). Detecting AI-generated code assignments using perplexity of large language models. *Proc. AAAI Conf. Artif. Intell.* 38, 23155–23162. doi: 10.1609/aaai.v38i21.30361

Xu, Z., Xu, R., and Sheng, V. S. (2024). Chatgpt-generated code assignment detection using perplexity of large language models (student abstract). *Proc. AAAI Conf. Artif. Intell.* 38, 23688–23689. doi: 10.1609/aaai.v38i21.30527

Zellers, R., Holtzman, A., Rashkin, H., Bisk, Y., Farhadi, A., Roesner, F., et al. (2019). Defending against neural fake news. *Adv. Neural Inf. Process. Syst.* 32.

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., et al. (2023). "CodeGeeX: a pre-trained model for code generation with multilingual evaluations on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (New York, NY: Association for Computing Machinery), 5673–5684. doi: 10.1145/3580305.3599790