An Embedding-Based Approach for Identifying LLM-Generated Code in Student Assignments

Paulina Gacek

AGH University of Krakow,
Faculty of Electrical Engineering, Automatics, IT and Biomedical Engineering,
Mickiewicza 30, 30-059 Krakow, Poland

Abstract

The widespread availability of large language models has introduced new challenges to academic integrity in programming courses. This paper presents a lightweight and practical system for detecting AI-assisted code submissions by leveraging code embeddings to compare student submissions against representative LLM-generated solutions. Experimental results from real student assignments in a university-level Algorithms and Data Structures course demonstrate that the system effectively highlights submissions with strong semantic similarity to LLM-generated solutions, even when minor edits were applied. This system can significantly reduce manual inspection workload by flagging suspicious submissions for instructor review, serving as a decision-support tool rather than an automated classifier.

Keywords

code similarity, large language models, AI-assisted plagiarism, code embeddings, programming education

1. Introduction

The emergence of large language models, such as GPT-4, has significantly reshaped computer science education. These models can solve complex algorithmic problems within seconds, pass technical interviews on platforms like LeetCode, and often outperform students in programming-related assessments [1]. While LLMs present valuable opportunities as educational tools, their use in academic settings raises serious concerns regarding academic integrity and the authenticity of student learning [2].

True coding proficiency develops through consistent engagement with algorithmic thinking, debugging, and mastering syntax. However, when students use large language models to generate solutions, they often bypass this process, hindering the development of fundamental programming skills. Despite these pedagogical concerns, students commonly employ LLMs for assignments [3].

This increasing reliance on AI in programming introduces unique detection challenges. While most current AI detection tools focus on AI-generated text, they largely overlook AI-generated code [4]. Distinguishing LLM-assisted or AI-generated code from human-written code is particularly complex. Unlike text detection, which can leverage stylistic features and linguistic patterns, programming tasks often have a constrained solution space [5]. Consequently, independently written solutions by different students might exhibit structural or semantic similarities, making it difficult to reliably differentiate between authentic and AI-generated code.

Building on the identified need, this paper introduces a lightweight and interpretable system designed to assist educators in identifying code submissions that may have been generated or significantly influenced by large language models. Rather than making grading decisions or issuing accusations, the system computes similarity scores between student submissions and representative LLM-generated solutions. These scores serve as a decision-support tool for educators, helping to flag submissions that necessitate a closer examination. Proposed system aims to foster informed, constructive dialogue between instructors and students, thereby promoting academic integrity in an era where LLMs are increasingly integrated into the programming process.

GeCoIn 2025: Generative Code Intelligence Workshop, co-located with the 28th European Conference on Artificial Intelligence (ECAI-2025), October 26, 2025 — Bologna, Italy
paulina.gacek.pl@gmail.com (P. Gacek)
0009-0008-1242-7542 (P. Gacek)

2. Methodology

The goal of the proposed system is to detect potential use of large language models in student programming assignments by simulating how students might realistically use AI-generated code while attempting to avoid detection. The approach combines LLM code generation with an embedding-based similarity analysis to compare student submissions against a range of plausible AI-generated solutions.

2.1. Modeling Student Behavior

To guide the design of the simulation, an anonymous survey was conducted among 50 Computer Science students at AGH University of Kraków. The results, summarised in Figure 1, indicated that 94% of students reported using large language models when solving programming tasks. Among these students, 66.7% stated they deliberately modify the generated code to avoid detection. Specifically, the most common modifications included removing comments, reported by 78.1% of LLM users who admitted to modify LLM-generated code. Additionally, 46.9% of LLM users reported changing variable names, and 46.9% made manual modifications to the code to make it appear less AI-generated, often involving structural or stylistic changes such as simplifying advanced constructions.

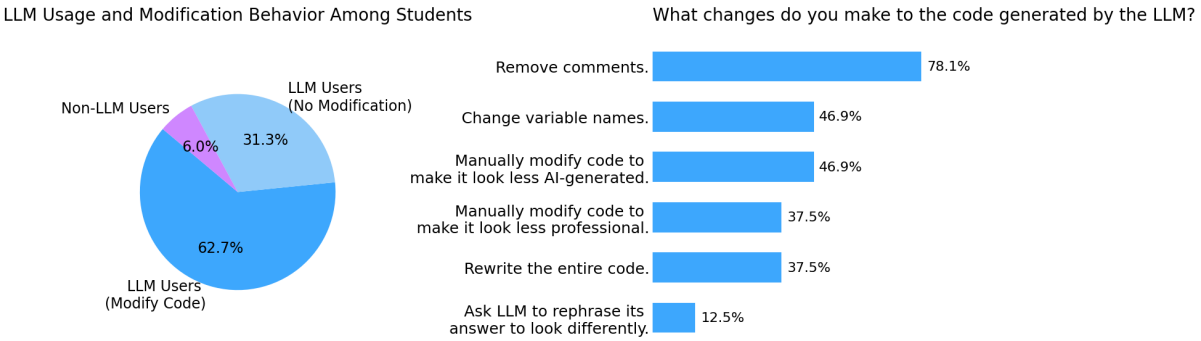

Figure 1: Distribution of LLM usage and code modification behaviors among surveyed students.

Complementing the findings on LLM adoption, survey also investigated the specific LLMs students utilize for programming tasks, as illustrated in Figure 2. The results indicate a predominant reliance on GPT, with 97.9% of students reporting its use. Other frequently employed LLMs include Deepseek (54.2%) and Gemini (41.7%), while models such as Claude (20.8%), Grok (8.3%), Copilot (2.1%), Llama (2.1%), and Mistral (2.1%) were used to a lesser extent. This distribution highlights the landscape of tools students are currently integrating into their programming education.

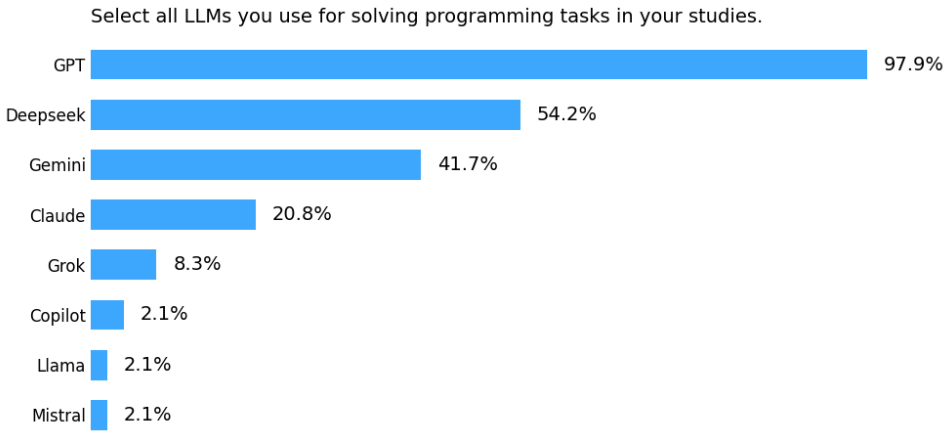

Figure 2: Distribution of Large Language Models used by surveyed students for programming tasks.

2.2. Code Generation and Augmentation

The system begins by querying selected large language model to generate a reference solution for a given programming assignment. To simulate realistic student behavior, the model is then prompted to rewrite the code k times while preserving its original functionality. Each rewritten version represents a variation that a student might plausibly submit. This process is illustrated in Figure 3.

A key architectural advantage of proposed system is its extensibility, allowing for easy integration of additional LLM API connections. In the current implementation, based on the prevalent usage observed in the survey, APIs to GPT-4o[1] and Gemini 2.5 Flash[2] have been incorporated.

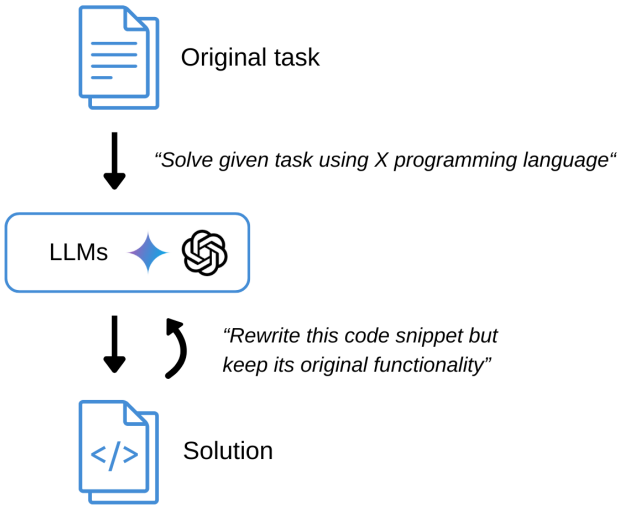

Figure 3: Illustration of the code generation and rewriting process using an LLM.

The output of this step is a set of diverse yet semantically equivalent code snippets that reflect the kinds of edits students might apply to LLM-generated code before submission.

2.3. Similarity Detection Using Embeddings

To assess whether a student submission may be derived from an LLM-generated solution, an embedding-based similarity detection approach was employed, as illustrated in Figure 4.

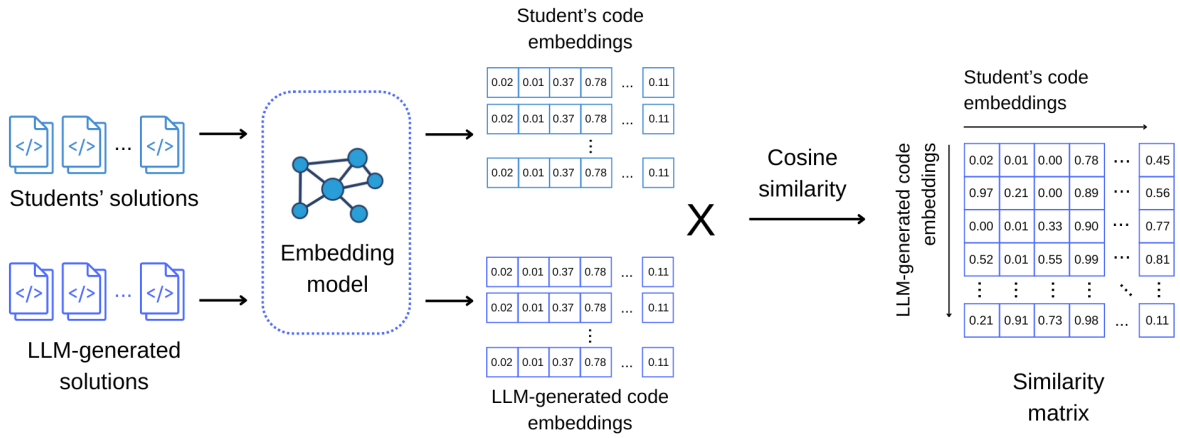

Figure 4: Overview of the embedding-based similarity detection pipeline.

[1] https://openai.com/index/hello-gpt-4o/
[2] https://deepmind.google/models/gemini/flash/

The process begins by removing all comments from both student submissions and LLM-generated code variants to eliminate stylistic noise. The cleaned code snippets are then embedded using Qodo-Embed-1 [6], a state-of-the-art embedding model trained on a diverse corpus of programming code, including Python. This model captures the semantic structure of code and is robust to minor textual variations, such as changes in variable names or formatting.

Cosine similarity is then computed between each student submission and each LLM-generated variant. This yields a quantitative measure of semantic similarity, allowing the system to identify structurally or algorithmically similar code even when superficial edits are present. High similarity scores may suggest that a student submission was influenced—either directly or indirectly—by AI-generated content.

It is important to emphasize that similarity scores alone do not provide definitive evidence of misconduct. Due to the narrow solution space of many programming problems, different students may independently arrive at similar implementations. Consequently, no fixed threshold can guarantee perfect discrimination between original and AI-assisted submissions. In this study, the similarity cutoff was empirically set to 0.94, based on manual analysis of the submission dataset and the observed distribution of similarity scores.

This embedding-based approach significantly reduces the manual workload associated with reviewing large numbers of student submissions. In scenarios involving over a hundred code submissions, detailed manual inspection is often impractical. The system efficiently identifies potentially AI-influenced code, enabling instructors to concentrate their efforts on a small subset of cases that warrant further investigation.

3. Experiments

To assess the effectiveness of the proposed system in detecting AI-assisted student submissions, a series of experiments were conducted using real-world programming assignments and authentic student data. The evaluation was performed on a dataset comprising 509 student submissions collected across multiple assignments from the Algorithms and Data Structures course at AGH University of Krakow. Each task required students to implement a function that solves a well-defined algorithmic problem, applying data structures and techniques introduced during the course.

Evaluating AI-generated code detection is inherently challenging due to the absence of definitive ground truth labels. In typical academic settings, students do not disclose whether they have used large language models to assist with their work, making it difficult to construct a reliably labeled dataset.

While one possible solution would involve instructing students to use LLMs and intentionally disguise the resulting code, this approach risks creating an artificial environment. When participants are aware their submissions will be analyzed for AI traces, they may overcompensate in their modifications—potentially expending as much effort on concealment as they would on solving the problem unaided. As a result, the behavior captured in such a setup may not accurately reflect real-world scenarios and could lead to skewed conclusions.

To address these challenges without introducing artificial bias, we adopt an unsupervised evaluation approach. The system generates multiple LLM-based variants of each programming task and cosine similarity is then computed between student submissions and these generated variants. Rather than relying on binary labels, the distribution of similarity scores across all submissions is analyzed. Submissions with high similarity scores are interpreted as likely influenced by LLM-generated content.

3.1. Case Study: Example Assignment Analysis

To further illustrate the effectiveness of the proposed detection system, this section analyzes a representative programming task used in the evaluation. The full task description is provided below.

Task description

AGH student city is covered with trees that have an extensive root system. This system is represented by a graph G, where vertices represent trees and edges represent connections between their root systems. To study the city's root system, students selected k trees and inoculated them with k different fungus species, numbered from 0 to $k-1$.
In one unit of time, a fungus can spread from a tree to all directly connected trees whose roots were not previously infected by any fungus. If two or more fungus species reach an uninfected tree in the same unit of time, the fungus with the smallest index wins and infects that tree.
The task is to implement the function getCountOfInfectedTrees(G: List[List[int]], infectedTrees: List[int], fungusNr: int) -> int, which determines how many trees will ultimately be infected by the fungus with the number fungusNr. The function accepts the following arguments:

- G: The graph represented as an adjacency list.
- infectedTrees: An array containing the numbers of the trees that were initially inoculated with fungus.
- fungusNr: The number of the fungus for which we want to count infected trees.

The function should return the number of trees infected by fungus number fungusNr.

Figure 5 presents the distribution of cosine similarity scores calculated between 102 student submissions and a set of $k = 4$ LLM-generated solutions for the above task using GPT-4o and Gemini Flash 2.5 APIs.

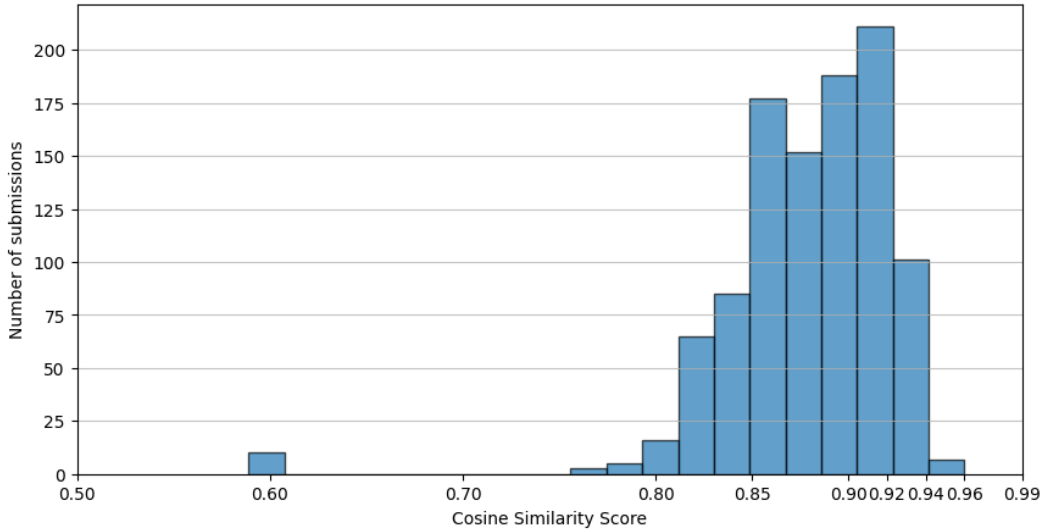

Figure 5: Distribution of pairwise cosine similarities between student submissions and LLM-generated solutions for the example assignment.

The majority of submissions exhibit cosine similarity scores in the range of 0.84–0.94, indicating that many student solutions share structural or semantic similarities with LLM-generated code. A small number of submissions exceed a similarity of 0.94, suggesting a strong resemblance that may indicate AI assistance, despite superficial variations. Interestingly, the ten lowest scores (around 0.60) represent comparisons between LLM-generated answers and a single student submission containing only the function definition without implementation.

Listings 1 and 2 present a student submission and its corresponding GPT-4o-generated solution, which yielded a cosine similarity score of 0.96. Although these implementations differ slightly in control flow and formatting, both employ a nearly identical breadth-first search strategy to simulate the spread of fungal infection across the graph.

```
def getCountOfInfectedTrees(G: List[List[int]], infectedTrees: List[int],
                                                fungusNrToCount: int) -> int:
    n = len(G)
    owner = [-1] * n
    time = [-1] * n
    queue = deque()
    for fungus_id, tree in enumerate(infectedTrees):
        owner[tree] = fungus_id
        time[tree] = 0
        queue.append((tree, fungus_id, 0))
    while queue:
        node, fungus_id, t = queue.popleft()
        for neighbor in G[node]:
            if owner[neighbor] == -1:
                owner[neighbor] = fungus_id
                time[neighbor] = t +1
                queue.append((neighbor, fungus_id, t + 1))
            elif time[neighbor] == t + 1 and fungus_id < owner[neighbor]:
                owner[neighbor] = fungus_id
    return owner.count(fungusNrToCount)
```

Listing 1: Student submission with 96% cosine similarity to an GPT-4o-generated variant.

```
def getCountOfInfectedTrees(G: List[List[int]], infectedTrees: List[int],
                                                fungusNr: int) -> int:
    n = len(G)
    owner = [-1] * n
    queue = deque()
    for fungusIndex, tree in enumerate(infectedTrees):
        owner[tree] = fungusIndex
        queue.append((0, fungusIndex, tree))

    while queue:
        time, fungusIndex, currentTree = queue.popleft()

        for neighbor in G[currentTree]:
            if owner[neighbor] == -1:
                owner[neighbor] = fungusIndex
                queue.append((time + 1, fungusIndex, neighbor))
    return owner.count(fungusNr)
```

Listing 2: Corresponding GPT-generated solution after 4th rephrasing (comments removed).

A notable observation is the reuse of specific variable names—such as `owner`—which are not particularly intuitive for the task at hand. The repeated use of such an idiosyncratic identifier in both versions suggests a high likelihood of copying or direct influence from the LLM output. This example demonstrates the strength of embedding-based similarity detection in capturing semantic and structural similarities that go beyond surface-level modifications.

Importantly, the system does not make automatic accusations or grading decisions. Instead, it provides similarity metrics to support educators in reviewing potential AI-assisted solutions and initiating conversations with students if necessary.

In this particular case, an analysis of the student's submission history reveals additional evidence suggesting the use of a large language model. The entire implementation was submitted within a span of just 10 minutes and triggered two critical errors on first two submissions: `ERROR: name 'deque' is not defined` and `ERROR: name 'fungusNr' is not defined`. These errors are typical when copying code from an LLM response without adapting it to the provided function signature or including necessary imports. Notably, the function argument `fungusNr` was incorrectly replaced by `fungusNrToCount`, further supporting the hypothesis that the submission was pasted from a generic LLM output without adequate integration into the student's codebase.

In contrast, the solution shown in Listing 3 was confirmed to be independently written by a student without the use of AI assistance. This conclusion was based on direct communication with the student and a review of their submission history. While the core logic of both this and the suspected AI-assisted solutions relies on breadth-first search (BFS), notable differences exist in completeness, variable naming, and implementation structure. These distinctions are sufficient to account for the lower similarity score observed.

```
def getCountOfInfectedTrees(G: List[List[int]], infectedTrees: List[int],
                                    fungusNrToCount: int) -> int:
    discovered = [False] * len(G)
    queue = deque()
    f_counters = [0] * len(infectedTrees)
    for i in range(len(infectedTrees)):
        queue.append((infectedTrees[i], i))
        f_counters[i] += 1
        discovered[infectedTrees[i]] = True
    while queue:
        node, f_index = queue.popleft()
        for neighbour in G[node]:
            if discovered[neighbour] == False:
                discovered[neighbour] =
    return f_counters[fungusNrToCount]
```

Listing 3: Student-created solution confirmed to be written independently

As shown in Figure 6, there is a clear separation in cosine similarity scores between independently written code and submissions suspected to have been AI-assisted. Original work clustered in the 0.79–0.85 range, while potentially AI-generated submissions exhibited significantly higher similarity, between 0.90 and 0.96. This supports the validity of the similarity threshold selected and demonstrates the effectiveness of the embedding-based approach in flagging suspicious cases.

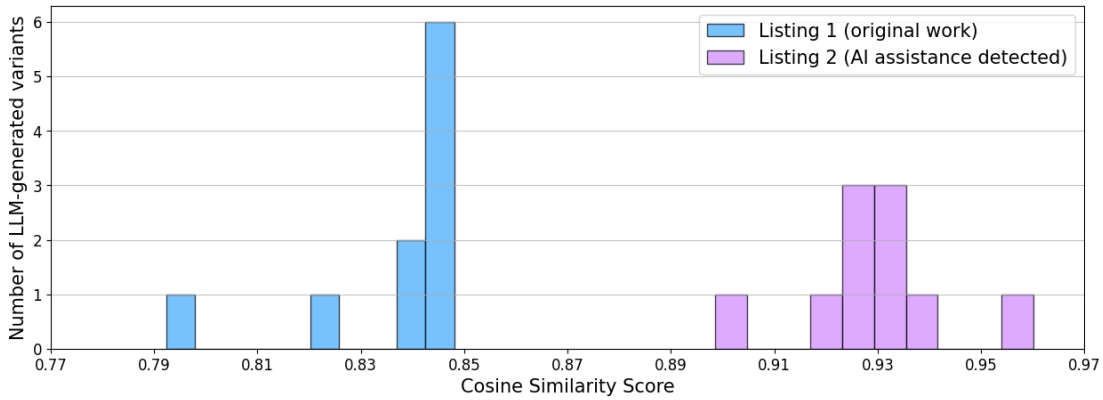

Figure 6: Distribution of cosine similarity scores between students submissions and LLM-generated variants.

4. Conclusion

This study presents a practical and scalable approach to detecting LLM-assisted code submissions in programming assignments. Experimental results on real-world student submissions from a university-level Algorithms and Data Structures course demonstrate the system's ability to capture both structural and semantic similarities, even when minor surface-level changes are introduced.

The proposed system avoids the need for ground-truth labeling, which is inherently difficult to obtain. Instead, it provides a data-driven, unsupervised methodology for supporting educators in evaluating the integrity of programming assignments. Instructors can use this tool to prioritize manual review of highly similar submissions, engage students in follow-up discussions, and better understand how LLMs are influencing learning behaviors. For example, the identification of a student who submitted an entire solution within minutes—along with LLM-typical syntax and missing imports—suggests that detection systems can serve as valuable starting points for personalized pedagogical intervention.

5. Future Work

In the current approach, only the final version of each student submission is used for similarity computation. Future iterations will take into account all saved submissions, allowing the detection method to capture the evolution of problem-solving and style shifts across attempts. In addition, incorporating other types of similarity is planned in order to improve the robustness of the approach.

A. Source Code Availability

The source code and supporting materials used in this study are openly available in a public GitHub repository: https://github.com/paulinagacek/llm-detection-gecoin.

Declaration on Generative AI

During the preparation of this work, the author used GPT-4o and Gemini 2.5 Flash in order to: Grammar and spelling check. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, Y. Zhang, Sparks of artificial general intelligence: Early experiments with gpt-4, 2023. URL: https://arxiv.org/abs/2303.12712. arXiv:2303.12712.

[2] B. Qureshi, Chatgpt in computer science curriculum assessment: An analysis of its successes and shortcomings, in: 2023 9th International Conference on e-Society, e-Learning and e-Technologies, ICSLT 2023, ACM, 2023, p. 7–13. URL: http://dx.doi.org/10.1145/3613944.3613946. doi:10.1145/3613944.3613946.

[3] T. Paustian, B. Slinger, Students are using large language models and ai detectors can often detect their use, Frontiers in Education Volume 9 - 2024 (2024). URL: https://www.frontiersin.org/journals/education/articles/10.3389/feduc.2024.1374889. doi:10.3389/feduc.2024.1374889.

[4] Z. Xu, V. S. Sheng, Detecting ai-generated code assignments using perplexity of large language models, Proceedings of the AAAI Conference on Artificial Intelligence 38 (2024) 23155–23162. URL: https://ojs.aaai.org/index.php/AAAI/article/view/30361. doi:10.1609/aaai.v38i21.30361.

[5] R. Azoulay, T. Hirst, S. Reches, Let's do it ourselves: Ensuring academic integrity in the age of chatgpt and beyond, 2023. doi:10.36227/techrxiv.24194874.v1.

[6] Q. AI, Qodo documentation, 2024. URL: https://docs.qodo.ai/qodo-documentation, accessed: 2025-06-08.