Interactive Evaluation of Complex Programming Assignments Using LLM Assistant

Tomáš Kormaník
Department of Computers and Informatics, FEI TU of Košice, Slovakia

Viktória Lukáčová
Department of Computers and Informatics, FEI TU of Košice, Slovakia

Jaroslav Porubän
Department of Computers and Informatics, FEI TU of Košice, Slovakia

Abstract

Generative language models present significant advancements in artificial intelligence with increasing applications in software engineering education. This paper explores the potential of customized generative dialogue models for automated assessment of programming assignments. The research introduces *KP Assistant*, a tailored implementation based on GPT-4o developed for a *Component Programming* university course. The research evaluated the effectiveness of this approach in generating relevant questions about source code, assessing student understanding, and providing objective feedback through a series of experiments with various game implementations and student testing. The findings demonstrate the feasibility of integrating such models into educational workflows while also acknowledging their present limitations. The study provides a framework for implementing similar systems in programming education, showing how generative AI can augment traditional assessment methods while maintaining pedagogical integrity.

2012 ACM Subject Classification General and reference → Empirical studies; Software and its engineering → Empirical software validation; Computing methodologies → Natural language generation; Social and professional topics → Student assessment

Keywords and phrases Artificial Intelligence, Generative Models, Programming Assessment, Software Engineering Education

Digital Object Identifier 10.4230/OASIcs.ICPEC.2025.13

Funding This work was supported by project KEGA No. 061TUKE-4/2025 Building Bridges between University and High School ICT Education.

1 Introduction

The emergence of large language models (LLMs) recently has transformed various domains, including software engineering and education. These models, trained on vast amounts of textual data, demonstrate remarkable capabilities in understanding and generating human-like text, including programming codes. With the introduction of conversational interfaces like ChatGPT, Claude, and others, a new paradigm of human-AI interaction has emerged – generative dialogue.

In software engineering education, the assessment of programming assignments has traditionally been a time-consuming and sometimes subjective process. Instructors must evaluate not only the functionality of student-submitted code but also assess the student's understanding of underlying concepts, design choices, and implementation details. This multifaceted evaluation process presents an opportunity for generative dialogue models to assist, potentially reducing faculty workload while maintaining or even enhancing assessment quality.

The findings presented in this paper contribute to the growing body of knowledge about AI applications in education, particularly in the context of programming assessment. The study demonstrates both the promising capabilities and current limitations of this approach, offering helpful feedback to educators and researchers interested in leveraging generative AI for educational purposes.

2 Generative Language Models in Education

Recent studies have demonstrated the potential of generative AI in educational contexts. In their 2023 study, Yilmaz and Karaoglan Yilmaz [8] examined the impact of generative AI tools on students' computational thinking skills, programming self-efficacy, and motivation in an object-oriented programming course. Their experiment involved 45 undergraduate computer science students divided into experimental and control groups. The results showed significant improvements in computational thinking, particularly in creativity (f = 0.427)[1], algorithmic thinking (f = 0.163), collaboration (f = 0.102), critical thinking (f = 0.102), and problem-solving abilities (f = 0.151). Students in the experimental group also demonstrated higher programming self-efficacy (f = 0.265) and overall motivation (f = 0.214).

Gouia-Zarrad and Gunn [4] explored the use of ChatGPT in a differential equations course. Their research involved 90 engineering students who used ChatGPT for numerical implementations of differential equations. The findings revealed that 91% of students reported improved programming skills, 72% noted better understanding of numerical implementations, and 70% indicated that ChatGPT helped them solve specific challenges. The study highlighted ChatGPT's effectiveness as a programming assistant, particularly in providing personalised help tailored to each student's knowledge level.

Our department is focusing on the evaluation of source codes extensively, whether it is at non-LLM automated solutions or whole systems designed to test whole projects written in different languages. One of our researchers, Marek Horváth, focuses on the analysis and detection of LLM-generated source codes and has created several works that help us significantly. Even though his work is not focused on the evaluation of source codes using LLMs, various information can be learned about how source code is understood by LLMs and what their limitations are. His work regarding variants of C programs generated by ChatGPT[6] demonstrated to us how LLMs approach each prompt and what the results are. On the contrary, he also explores possibilities to compare resulting source codes, recently by binary decomposition of resulting solutions[7]. Naturally, there are more researchers available at our department, but they focus mainly on standard evaluation methods based on algorithms and comparative analysis.

2.1 Code Generation and Analysis

The capabilities of LLMs in code generation and analysis have been explored extensively. Several systems like GitHub Copilot, which utilizes OpenAI's Codex and GPT-4 models, have demonstrated impressive performance in generating functional code blocks based on natural language prompts or existing project context [2]. These systems integrate into popular development environments and assist programmers in various tasks, including code generation, refactoring, error identification, and documentation.

[1] Cohen's f ratio – measure of the strength of the relationship between variables

Research by Ziegler et al. [9] on productivity assessment of neural code completion tools found significant improvements in developer performance, particularly for routine programming tasks. The integration of these tools into programming workflows has shown potential for enhancing both efficiency and code quality.

3 Methodology

3.1 Design and Implementation

KP Assistant was developed as a customised GPT (Generative Pre-trained Transformer) using OpenAI's GPT Builder platform. This approach allowed for precise configuration of the model's behaviour through specialised instructions, knowledge base integration, and capability settings. The underlying model selected was GPT-4o due to its superior performance in code understanding and analysis during preliminary testing compared to alternative models.

An important characteristic of *KP Assistant* is that it was fully configured to operate in the Slovak language, with all instructions, prompts, and interactions designed specifically for Slovak-speaking students. This language specification was critical for seamless integration into the Technical University of Košice's curriculum, where Slovak is the primary language of instruction. However, some leeway was provided in terms of in-code comments or technical terms, which are by industry best practices in the English language.

Based on previous research at our department and other universities, we were able to design clear instructions for the GPT model in terms of expected answer structure and overall GPT's persona. Numerous sources provide recommendations for teachers and instructors in undergraduate and graduate study programmes. These are usually based on the observed behaviour and work results of students. One of the largest and more recent studies with well documented results is written by García et al.[3], where they conclude that informal and more open behaviour leads to better results. Özdaş [10] came to a similar conclusion, along with a list of key characteristics of a positively viewed educator, which were easy to convert to character traits of our GPT. Configuring our GPT's behaviour based on these and other relevant findings created a hypothetically ideal persona for our assistant.

The assistant was configured with specific instructions divided into three main categories:

- **Basic Instructions:** Defined the overall purpose, interaction style, and operational parameters.
- **Conversation Start Instructions:** Established fixed prompts for initiating different types of assessment sessions.
- **Question Generation Instructions:** Detailed guidelines for analysing student code, generating relevant questions, and evaluating responses.

These instructions were not provided all at once to the assistant but were served behind the scenes at predetermined phases of conversation. The whole conversation flow is visualized in the provided diagram (Fig. 1), which shows expected input and action based on it. Simply said, these instructions were served if previous input was deemed valid and sufficient for further processing, which streamlined the process of the whole conversation. Additionally, in this case we personally witnessed that hallucinations from GPT were more scarce, since it was validating itself. Naturally, this more complex design made conversation slightly slower, mainly after users uploaded their assignment. However, delay after this step was previously present as well, and the time difference was deemed insignificant (approx. 10 seconds longer in 10 testing runs).

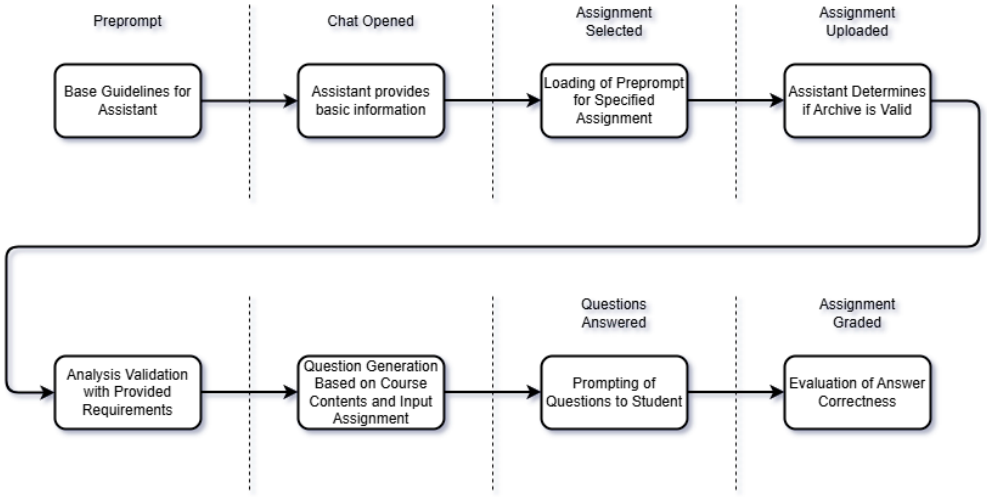

Figure 1 Flow of interaction with *KP Assistant* with valid inputs from user/student.

The assistant's knowledge base was populated with course materials, including lecture presentations and descriptions of game assignments that students could implement. This provided essential context about the course structure, learning objectives, and assignment requirements. Web search and code interpretation capabilities were enabled to allow access to the course website and facilitate source code analysis.

Three conversation starters were defined corresponding to the three assignment submission phases in the Component Programming course:

- **Submission 1:** Game logic and JDBC
- **Submission 2:** JPA + REST
- **Submission 3:** Web browser game with GUI

For each submission type, the assistant was configured to generate and assess five questions based on the submitted code: two easy questions (3 points each), two medium-difficulty questions (5 points each), and one difficult question (6 points). Additionally, the assistant was programmed to evaluate specific aspects of the implementation based on course criteria, such as game functionality, service component integration, code quality, and testing.

3.2 Testing with Diverse Game Implementations

Six different game implementations were used to assess the assistant's performance across various code structures and quality levels: Battleships, Lights Off, Color Sudoku, NumberLink, Reversi, and Connect Four. These implementations varied in complexity, code organization, and overall quality, providing a comprehensive test bed for the system.

For each game, a structured testing procedure was designed and conducted:

- Initiating a conversation using one of the submission options
- Uploading the game's source code as a ZIP archive
- Answering the generated questions
- Analysing the final assessment and score
- Testing various edge cases and unexpected behaviours

3.3 Student Evaluation

A total of 16 students from the Component Programming course participated in testing the system, with 12 providing valid feedback. Students were instructed to follow provided steps in this order:

1. Access *KP Assistant* using a provided link
2. Select the appropriate submission – in this case, first submission option
3. Upload their own game implementation
4. Answer the generated questions
5. Complete a feedback survey evaluating their experience

The survey collected both quantitative assessments (on a 5-point scale) of conversation flow, question relevance, clarity, and evaluation objectivity, as well as qualitative feedback on perceived strengths and weaknesses of the system.

3.4 Data Collection and Analysis

Data was collected from both experimental phases, encompassing conversation logs from the system tests with the six reference games, student-assistant interaction transcripts, and comprehensive survey responses from student participants.

The analysis methodology centred on identifying recurring patterns in three critical areas: the quality of question generation, the accuracy of assessment procedures, and any usability challenges that emerged during testing. Evaluation criteria were specifically designed to measure the relevance of generated questions to the actual submitted code, verify assessment consistency against expected scoring benchmarks, evaluate the natural flow and clarity of conversations, detect instances of hallucination where questions referenced nonexistent code elements and gauge overall student perception of the system through their qualitative feedback.

4 Evaluation of Results

We have decided to evaluate results for each game individually since the output of LLMs is based on how well they are trained, so it is likely that the original training data contained different amounts of information about each game, which can affect results. Evaluation from the student's point of view is also needed since their grades are going to be affected by this system.

4.1 Performance with Reference Games

Testing with the six reference game implementations revealed several key insights about *KP Assistant*'s performance:

- **Battleships:** The assistant performed well with this well-structured implementation, correctly analysing the code structure and generating relevant questions. The final assessment (33/34 points) aligned with the expected score based on code quality and response accuracy.
- **Lights Off:** During this test, the system was deliberately challenged by first uploading incorrect file formats and then a valid but unrelated ZIP file. The assistant correctly identified these issues and prompted for a proper submission. When the actual game code was uploaded, one hallucination occurred where the assistant asked about a nonexistent

method. After being informed of this error, it generated a replacement question, but failed to maintain the correct question sequence, skipping directly to the fifth question. The final score (24/34) appropriately reflected the code quality and response accuracy.

- **Color Sudoku:** The assistant generated generally relevant questions, but one hallucination occurred. Interestingly, when provided with a response that pointed out the hallucination, the assistant accepted the answer as correct and awarded full points. This behaviour indicates a potential evaluation vulnerability when students identify system errors. During further testing, we were not able to replicate this behaviour in 50 attempts with the same input, but its singular occurrence proves that even when analysing a fairly common game, the LLM cannot be completely trusted.
- **NumberLink:** When testing different submission types, the assistant could successfully transition between them upon request. However, when challenged about point deductions, the assistant occasionally displayed flexibility in scoring that could potentially be manipulated, suggesting the need for stricter evaluation protocols.
- **Reversi:** This test involved a large codebase (42MB ZIP archive), which revealed limitations in the assistant's code analysis capability. Examination of the code analysis logs showed that the assistant only processed the first ten files in the archive, leading to incomplete understanding of the codebase and subsequent hallucinations in questions.
- **Connect Four:** This implementation was intentionally minimal and low quality. The assistant correctly awarded low scores (3 points) for the basic aspects, reflecting the poor implementation. When asked to generate a complete working implementation of the game, the assistant appropriately refused and instead offered guidance on resources and documentation that would help improve the implementation.

4.2 Student Evaluation

The student evaluation provided valuable insights into the practical application of *KP Assistant* in an educational setting. The distribution of game implementations among the 12 valid student responses. We had additional games available among our submitted assignments, which allowed us to test the versatility of *KP Assistant* in scenarios that were not mentioned in the provided materials. Additional games included *Boulder Dash*, *Pipes*, *Nonogram*, *Bricks Breaking*, *Dots*, *Kakuro*, *Checkers*, and *Slitherlink*. Two of these games, *Dots* and *Kakuro*, were made by two individual students, which allowed us to observe grading differences among students with the same assignment but slightly varying results.

Students rated the **conversation flow and clarity** highly, with an average score of 4.42/5. Eight students gave the highest rating (5 – excellent), one rated it as excellent (4), and three rated it as good (3). This suggests that the assistant's communication style and instructions were generally clear and easy to follow. The **relevance of generated questions** to the actual code submitted received a moderate average rating of 3.42/5. Distribution was more varied, with three excellent ratings, three very good, three good, two below average, and one poor. This metric correlates with the hallucination issues observed in the reference game testing, indicating that question generation based on code analysis remains a challenging aspect of the system. **Clarity and unambiguity** aspect received an average rating of 3.58/5, with two excellent ratings, six very good, two good, one below average, and one poor. The assistant's ability to formulate clear, grammatically correct questions in Slovak was generally successful. Students **perceived the final assessment as relatively objective**, with an average rating of 4/5. Five students rated it as excellent, four as , one as good, and two as below average. This indicates that despite some issues with question generation, the overall evaluation framework generally aligned with student expectations of fair assessment.

4.3 Key Takeaways and Potential Applications

Among many takeaways, most obvious was the conversational approach which allowed for a more engaging and natural assessment experience compared to traditional automated testing systems. Students appreciated the clear communication style and the opportunity to explain their implementation decisions. *KP Assistant* demonstrated the ability to analyze various game implementations and generate relevant questions about their specific structures and features. This adaptability represents a significant advantage over rule-based assessment systems that typically require separate configuration for each assignment type.

The system successfully integrated both question-based assessment of student understanding and automated evaluation of code quality and functionality aspects. This holistic approach aligns well with educational best practices that emphasize both product and process evaluation. Unlike traditional office hours or scheduled defenses, the assistant remains available 24/7, allowing students to practice and prepare for formal assessments at their convenience. This could be particularly valuable for distance learning scenarios or courses with large student populations. Well structured prompts allowed for relatively straightforward adaptation to different courses, programming languages, and assessment criteria. This versatility suggests potential for broader application across the computer science curriculum.

4.4 Limitations and Challenges

Despite its promising capabilities, several limitations emerged during the evaluation of *KP Assistant*. The most significant technical limitation observed was the assistant's **restricted ability to process and understand large codebases** comprehensively. The processing of only the first ten files in larger projects led to incomplete code understanding and subsequent hallucinations. This suggests that improvements in code indexing and analysis are necessary for handling complex projects. The **occurrence of questions about nonexistent code elements** represents a critical challenge for educational applications. Such hallucinations could confuse students and potentially undermine their confidence in both their code and the assessment system. The assistant's **tendency to accept explanations about hallucinated questions** as correct answers indicates a potential vulnerability that could be exploited. A more robust verification mechanism is needed to maintain assessment integrity. The most effective mitigation of this issue was the implementation of carefully crafted assignments which were designed for this use case. To utilise LLM assistant with the best results, we tailored specific assignments (games in our case) specifically with future analysis by LLM assistant in mind. We suggest creating these assignments with a maximum of 8 files in mind, since students can potentially create their own documentation or environment files.

From architectural perspective, the observed behaviour changes following an OpenAI platform update highlighted the **potential instability** of systems built on third-party AI services. Educational applications require consistency and reliability that may be challenging to maintain with rapidly evolving commercial AI platforms. The current implementation also **requires manual ZIP file uploads** rather than direct integration with development tools or submission systems, creating friction in the assessment workflow. This can be eliminated by switching to locally deployed models, which can be integrated into complex pipelines, but at the cost of convenience for both user and developer.

While students can freely access and use the custom GPT without a paid subscription, creating and maintaining such customized assistants **requires a paid OpenAI subscription for developer**. This introduces an ongoing cost consideration for educational institutions looking to implement and maintain similar systems long-term.

4.5 Integration into Educational Practice

KP Assistant and similar systems should serve as supplements to, rather than replacements for, instructor evaluation. Their primary value lies in providing **additional practice opportunities** and preliminary feedback before formal assessment. For the Component Programming course specifically, the assistant could be utilized as a pre-defense preparation tool, allowing students to identify potential weaknesses in their implementation before the official submission. Implementation **should include mechanisms for instructor oversight**, such as conversation logging and the ability to review and override automated assessments when necessary. In the context of Technical University of Košice's curriculum, such features could be implemented through the existing GitLab system, where conversation logs could be stored alongside student submissions for reference during final evaluation.

We should inform students about the system's limitations to set appropriate expectations and encourage critical evaluation of the assistant's feedback. The documentation for the Component Programming course could include specific guidance on how to interpret and critically evaluate *KP Assistant*'s responses, particularly regarding the known hallucination issues with complex codebases. Regular updates to the system's knowledge base and instructions based on observed performance and student feedback are essential for maintaining relevance and accuracy. We should refresh lecture materials and game descriptions in the assistant's knowledge base for each semester to ensure alignment with current course content. The experience from one academic year could inform improvements for the next.

Combining dialogue-based assessment with traditional automated testing could leverage the strengths of both approaches while mitigating their respective weaknesses. The Java-based projects in the Component Programming course already include unit tests, which could be complemented by *KP Assistant*'s understanding-focused questions to provide a comprehensive assessment of both code functionality and student comprehension. This implementation enhances comprehension by providing Slovak language support. Maintaining proper localization is crucial for educational applications targeted at large groups of students. Future iterations could strengthen the Slovak technical vocabulary used in programming contexts, ensuring that terminology is consistent with that used in lectures and course materials at Slovak universities.

Ideally, phased implementation beginning with lower-stakes assignments before expanding to critical assessments would allow both students and faculty to adjust to this new assessment paradigm. For example, *KP Assistant* could first be used only for the initial submission in the Component Programming course before potentially expanding to later submissions. Because of the adaptable design shown with *KP Assistant*, we could create similar tools for other programming classes, ensuring a uniform assessment experience throughout the computer science program while still meeting the unique needs of each class. This claim is supported by the fact that our assistant was able to analyze the source codes of games that were not included in the provided materials but had similar mechanisms and components inside.

We utilised our previously existing bundle of solutions to evaluate student assignments – ARENA. This system has been developed over the last 5 years, and it has become part of half a dozen subjects in our department. Further details can be viewed in the previous volume of ICPEC[5], which also refers to the previous versions of the system. One component of this system in particular, ORACLE, provides personalised feedback to the students, based on the runtime of their solutions, their coding practices and their way of designing the algorithms. Along with *KP Assistant*, it helps provide regular and relevant feedback to the students, which, as proven before, causes noticeable improvement in their performance. In

the future, we aim to create a self-hosted variant of *KP Assistant*, which can be deployed and implemented into the ARENA system. This will provide us valuable data about learning patterns, common issues, misunderstandings and mistakes. Adding *KP Assistant* to the ARENA system will allow the LLM to access student and reference source codes on the department's version control system, which will speed up the process of evaluation and simplify the process from the side of the students.

5 Future Work

Our focus will shift to improving code analysis capabilities. Hallucination issues can be addressed by creating assignments that are evaluated by *KP Assistant* and provided in various states of resolution. Developing tighter integration with existing educational technologies, either external or internal (such as ARENA), will provide more detailed insight into the education process for educators and researchers. As LLMs continue to advance, their application in programming education represents a promising frontier for enhancing both teaching efficiency and learning outcomes.

Furthermore, we see the potential of LLM Assistant when used on courses that are not focusing on evaluation of source code but on analysis of students approach to the successful solution of the problem. We performed pilot experiments with some assignments from the *Security of Computers and Computer Networks* course, where we supplied LLM with available log files, packet dumps (from either *tcpdump* or *Wireshark*), machine codes of software, and student evaluations of these indicators. LLM has processed these inputs and provided us with a grade and a worded evaluation of our approach.

Baráth and Turančík[1] are detecting select attacks with the MATLAB environment, which is effective in a production environment, but in an educational setting we can verify a larger sample of attacks and their preventions crafted by students at the cost of effectiveness, which is not relevant for us. Finally, this approach allows us to verify whether students understood the resources they mentioned, such as the previously discussed solution utilising ANN and classifiers, or if they misunderstood it, resulting in incorrect outcomes.

6 Conclusion

This study demonstrates the potential of customized generative dialogue models for automated assessment of programming assignments in software engineering education. *KP Assistant*, implemented using GPT-4o and tailored for a Component Programming course, showed promising capabilities in generating relevant questions, providing feedback, and evaluating student understanding across various game implementations.

The evaluation revealed both strengths and limitations of this approach. Key advantages include the interactive assessment format, flexibility across different implementations, and continuous availability. However, significant challenges remain, particularly regarding limited code analysis depth, hallucination issues, and evaluation vulnerabilities. The conversational nature of the LLM encouraged students to submit their assignments using this method or simply ask questions about their progress.

Our findings suggest that while generative dialogue models can serve as valuable educational tools, they are best positioned as supplements to, rather than replacements for, traditional assessment methods. With appropriate integration, oversight, and continuous refinement, such systems have the potential to enhance programming education by providing additional practice opportunities, reducing instructor workload for routine assessments, and offering students more immediate and comprehensive feedback.

References

1 Július Baráth and Michal Turčaník. Detection of selected attacks based on ann and classifiers. In *2023 Communication and Information Technologies (KIT)*, pages 1–4, 2023. doi:10.1109/KIT59097.2023.10297105.

2 Rishi Bommasani et.al. On the opportunities and risks of foundation models. In *Center for Research on Foundation Models (CRFM) at the Stanford Institute for Human-Centered Artificial Intelligence (HAI)*, July 2022. doi:10.48550/arXiv.2108.07258.

3 Alfonso Javier García, Facundo A Froment, and María Rocío Bohórquez. University teacher credibility as a strategy to motivate students. *Journal of New Approaches in Educational Research*, 12(2):292–306, 2023. doi:10.7821/naer.2023.7.1469.

4 Rim Gouia-Zarrad and Cindy Gunn. Enhancing students' learning experience in mathematics class through ChatGPT. *International Electronic Journal of Mathematics Education*, 2024. doi:10.29333/iejme/14614.

5 Marek Horváth, Tomáš Kormaník, and Jaroslav Porubän. Adaptation of Automated Assessment System for Large Programming Courses. In André L. Santos and Maria Pinto-Albuquerque, editors, *5th International Computer Programming Education Conference (ICPEC 2024)*, volume 122 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:11, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.ICPEC.2024.4.

6 Marek Horváth, Lenka Bubeňková, and Emília Pietriková. Exploring gpt-generated variations in c programming assignments. In *2025 IEEE 23rd World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 000331–000336, 2025. doi:10.1109/SAMI63904.2025.10883288.

7 Marek Horváth and Filip Gurbáľ. Code clones: A novel approach to detecting plagiarism in binary decomposition of c programs. *Acta Electrotechnica et Informatica*, 24(2):13–18, 2024. URL: https://www.proquest.com/scholarly-journals/code-clones-novel-approach-detecting-plagiarism/docview/3173384211/se-2.

8 Ramazan Yilmaz and Fatma Gizem Karaoglan Yilmaz. The effect of generative artificial intelligence (AI)-based tool use on students' computational thinking skills, programming self-efficacy and motivation. *Computers and Education: Artificial Intelligence*, 4:100147, 2023. doi:10.1016/j.caeai.2023.100147.

9 Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, May 2022. doi:10.48550/arXiv.2205.06537.

10 Faysal Özdaş. Teachers' immediacy behaviors and academic achievement: A relational analysis. *SAGE Open*, 12(2):21582440221091722, 2022. doi:10.1177/21582440221091722.