Too Hot To Handle: The Effects of Temperature on AI-Generated Code Used to Cheat in Programming Assignments

Robin Maisch*
robin.maisch@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Timur Sağlam*
timur.saglam@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Larissa Schmid
lgschmid@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Sebastian Hahner
sebastian.hahner@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Tobias Hey
tobias.hey2@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

In light of the rapid evolution of generative artificial intelligence, automatic code generation constitutes a novel cheating strategy for university programming courses. Varying the temperature parameter of LLMs, we generate nearly 10 000 programs for four programming tasks and present a comprehensive evaluation of their suitability for submission to mandatory assignments.

CCS Concepts

• **Software and its engineering**; • **Social and professional topics** → **Computer science education**; **Software engineering education**; • **Information systems** → *Near-duplicate and plagiarism detection*; • **Computing methodologies** → Artificial intelligence;

Keywords

Cheating, Academic Misconduct, Plagiarism, Software Plagiarism, Plagiarism Detection, Generative AI, Code Generation, Education

ACM Reference Format:
Robin Maisch, Timur Sağlam, Larissa Schmid, Sebastian Hahner, and Tobias Hey. 2026. Too Hot To Handle: The Effects of Temperature on AI-Generated Code Used to Cheat in Programming Assignments. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3774748.3795678

1 Problem Statement

Cheating on programming assignments is an ongoing challenge in university programming courses [11]. Students often obtain code for assignments from peers or online sources and reuse it in their own solutions, thereby undermining their own skill development. Educators rely on automated plagiarism detectors such as JPlag [5] to identify similarities among submissions even in large courses. To avoid detection by these tools, students *obfuscate* plagiarized code by modifying program features while preserving its functionality. Manual obfuscation approaches are common but largely ineffective [10]; automated approaches [2], specifically designed to avoid detection, are more effective, but countermeasures have also been thoroughly researched [7, 9].

New approaches arise with the rapid evolution of generative AI. Large Language Models (LLMs) such as ChatGPT can not only alter source code, but even provide complete solutions for textual assignment descriptions, thus making cheating easier than ever before [6]. As these tools are widely available and popular among students [4], they pose a severe challenge to academic integrity. However, it remains unclear whether solutions produced by current generative AI tools fully meet assignment requirements.

Early research indicates that, across multiple runs, generative AI produces code with a consistent structure, increasing risk of detection by plagiarism detectors when multiple students use the same language model [1]. However, existing works do not consider that students may alter LLMs' temperature settings to control creativity and indeterminism of the output. By generating programs at high temperatures, they may evade detection, thus threatening academic integrity. This raises broader questions about the feasibility of using AI-generated code as a cheating strategy in programming assignments, particularly given that temperature settings can significantly affect both the quality and variability of the output.

2 Research Questions

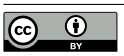
In this experiment, we examine the capabilities of generative AI to produce source code for university-level programming assignments as a novel approach to cheating. For four quality aspects, we assess both the overall performance and the influence of the temperature:

RQ1 Syntactical correctness (*measuring compilation rates*)
RQ2 Semantical correctness (*measuring test pass rates*)
RQ3 Structural similarity (*using automatic plagiarism detection*)
RQ4 Feasibility (*combining all three criteria in RQ1–3*)

3 Approach

To answer the research questions, we conduct a controlled experiment in which we generate and assess Java programs for four university-level programming assignments using the LLMs GPT-4.1, GPT-4o, and DeepSeek-V3 with varying temperature settings. For each combination of assignment, model, and temperature, the resulting programs are evaluated independently.

*Both authors contributed equally to this research.



ICSE-Companion '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3795678

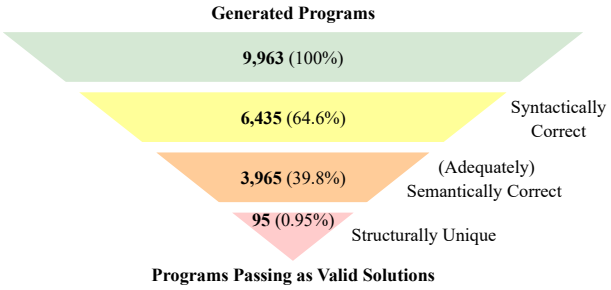


Figure 1: Generated programs that pass all evaluation stages.

Assignments. We use four assignments: *Sociology* and *Freckles* from the PROGpedia dataset [3] and *BoardGame* and *TicTacToe* from the introductory programming course at KIT [8]. These four assignments cover a wide range of complexity and program lengths.

Code Generation. For each assignment and model, we generate 30 programs at a temperature of 0.0, then increase in steps of 0.1 up to the maximum value of 2.0. We then compile all programs and collect those that fail. In subsequent iterations, we generate replacements for each incorrect program. We stop at the temperature value at which the compilation rate falls below 5 %. We employ zero-shot generation, never iterating on failing programs.

Evaluation. We then assess each program in four steps corresponding to the research questions, which we briefly describe.

Syntactic correctness We compile each of the nearly 10 000 AI-generated programs using `javac` from JDK 21.0.6. To pass this stage, programs must compile without errors.

Semantic correctness We run a set of predefined, task-specific functional tests on each compilable program. To pass this stage, programs must exceed a pass rate of 50 %.

Structural similarity Using JPlag [5], we assess compilable programs by task, model, and temperature, as well as the human solutions. Pairs of generated programs whose similarity exceeds that of human programs for the same task fail this stage.

Feasibility To assess the feasibility of passing programming assignments using zero-shot generative AI, we combine the results from the three previous stages.

4 Results

Syntactic correctness (RQ1) As shown in Figure 1, from a total of nearly 10 000 AI-generated programs, more than one-third contain syntactical errors. All three LLMs achieve high compilation rates at low temperatures, with a steady decline as temperature increases; at high temperatures, programs degrade severely. Interestingly, for DeepSeek, the decline is more gradual than for the GPT models, attributable to an internal downscaling of higher temperature values in DeepSeek's API. Also, the overall compilation rate is higher for the less complex assignments.

Semantic correctness (RQ2) About 40 percent of programs fail on the majority of test cases. Functional correctness is largely stable with respect to the temperature setting; however, across tasks and models, overall pass rate and variance differ notably.

Structural similarity (RQ3) For more than 99 % of generated programs, the best-matching program exceeds all pairs of human-written programs in similarity, thus drawing immediate attention. However, for the same configuration, non-similar program pairs also occur, often resulting in similarity ranges spanning 0–100 %.

Feasibility test (RQ4) Only 95 programs, 0.95 % of all generated programs, are inconspicuous *as well as* syntactically and semantically correct. These programs cover the whole configuration space of assignments, models, and temperatures.

5 Discussion and Conclusion

In this experiment, we generated nearly 10 000 programs for four university-level programming assignments using zero-shot LLM generation across different temperature settings and evaluated their quality. Across the entire temperature range, the effectiveness of cheating via generative AI was limited: almost all generated programs exhibited syntactic or semantic errors or showed a degree of structural similarity that would likely be flagged by automated plagiarism detection tools.

These findings do not indicate a general inadequacy of the evaluated models. Rather, they suggest that zero-shot code generation—a minimum-effort approach to cheating—is unlikely to yield passing solutions under the assumptions of our experimental setup. The effectiveness of more advanced strategies, such as prompt engineering, few-shot learning, or incremental code generation using more recent models, remains an open question.

Acknowledgments

This work was supported by the CHAINS project funded by the Swedish Foundation for Strategic Research (SSF), by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF), by the German Research Foundation (DFG) – SFB 1608 – 501798263, and by KASTEL Security Research Labs.

References

[1] Ashley Pang et al. 2024. ChatGPT and Cheat Detection in CS1 Using a Program Autograding System. *ITiCSE 2024*, 367–373. doi:10.1145/3649217.3653558
[2] Breanna Devore-McDonald et al. 2020. Mossad: Defeating Software Plagiarism Detection. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 138 (11 2020), 28 pages. doi:10.1145/3428206
[3] José Carlos Paiva et al. 2023. PROGpedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief* 46 (2 2023), 108887. doi:10.1016/j.dib.2023.108887
[4] Lilia Khendek et al. 2025. Exploring Data Science Students' Engagement, Usage Patterns, and Perceptions of Large Language Models in Programming. *Studies in Health Technology and Informatics* (2025). doi:10.3233/SHTI250547
[5] Lutz Prechelt et al. 2002. Finding plagiarisms among a set of programs with JPlag. *JUCS* 8, 11 (11 2002), 1016. doi:10.3217/jucs-008-11-1016
[6] Mohammad Khalil et al. 2023. Will ChatGPT get you caught? Rethinking of Plagiarism Detection. *Interacción* 14040 (2 2023). doi:10.48550/arXiv.2302.04335
[7] Robin Maisch et al. 2026. Same Same But Different: Preventing Refactoring Attacks on Software Plagiarism Detection. In *ICSE '26*. ACM NY, USA. doi:10.1145/3744916.3773225
[8] Timur Sağlam et al. 2024. Detecting Automatic Software Plagiarism via Token Sequence Normalization. *ICSE '24*, 113:1–113:13. doi:10.1145/3597503.3639192
[9] Timur Sağlam et al. 2025. Mitigating Obfuscation Attacks on Software Plagiarism Detectors via Subsequence Merging. In *ICSE '25 Companions: CSEE&T*. doi:10.5445/IR/1000179016
[10] Oscar Karnalim. 2016. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In *2016 ICTS*. IEEE, 63–68. doi:10.1109/icts.2016.7910274
[11] Chris Park. 2003. In Other (People's) Words: Plagiarism by university students–literature and lessons. *Assessment & Evaluation in Higher Education* 28, 5 (10 2003), 471–488. doi:10.1080/02602930301677