# Teaching and Using AI in the Classroom: Integrating AI Modules in Undergraduate Computer Science Education

Martin Cenek[1][0000-0001-7140-7084] and Ronnie Delos Santos[1]

University of Portland, Portland OR 97203, USA {cenek,delossan25}@up.edu

**Abstract.** Undoubtedly, the future workforce will be impacted by the increasing adoption of generative AI across industries. To prepare students for this evolving landscape, AI must be integrated into undergraduate computer science education. To address this, we explore the redesign of a core computer science course to include experiential learning modules on AI literacy, generative AI tools, and responsible AI usage. In addition, we report on the benefits, challenges, and ethical considerations of using small language models (SLMs) for automated grading. Finally, through an analysis of student engagement, learning outcomes, and perceptions of AI-driven instruction, we provide insights into best practices for responsible AI adoption in academia.

**Keywords:** teaching generative AI tools · AI skills for software engineers · generative AI in education.

## 1 Introduction

Over the last couple of decades, the science and practice of information retrieval have undergone fundamental changes. The transition from physical archives to digital repositories has made large amounts of information readily accessible. The use of search engines has gone from requiring knowledge of Boolean operators to supporting natural language processing (NLP). While this shift has increased accessibility, it has also changed how users interact with information. Today, users often pose poorly structured queries and receive equally suboptimal results due to the forgiving nature of modern search interfaces. A similar paradigm shift is now occurring with large language models (LLMs) and generative AI tools. These models enable users to generate content more efficiently, but their effective use heavily depends on users with AI literacy—an understanding of their mechanics, limitations, and appropriate use cases. Without structured guidance, students may over-rely on AI tools without proper evaluation or completely reject them because of uncertainty or ethical concerns. To proactively address this, we redesigned a core undergraduate computer science course to integrate experiential learning modules on generative AI literacy, responsible AI usage, and AI ethics [7]. These modules aim to teach students how generative AI models function, their practical applications, and the ethical implications of AI-generated content, particularly in relation to bias, fairness, and academic integrity [5].

Beyond AI literacy, another critical challenge is automated grading. Traditional grading methods are often subject to inconsistencies due to human bias and fatigue. As class size grows, manually evaluating student submissions becomes an increasing burden on instructors which may lead to delayed feedback for students. To alleviate this, automated grading systems have long been used to streamline assessment, but their effectiveness has been largely limited to structured formats like multiple-choice or numerical responses. While these approaches work well for rule-based evaluation, grading free-response answers presents a much greater challenge. Unlike structured questions, free-response grading requires not only identifying relevant keywords and syntax but also interpreting context, reasoning, and nuance. This complexity is further increased by the fact that students might make minor errors that humans intuitively recognize as acceptable, yet automated systems struggle to distinguish between a trivial mistake and a fundamentally incorrect response. As a product of this, achieving accurate and fair automated assessment of free-response answers requires more advanced techniques while minimizing misclassification and penalization.

In order to address these challenges, we present the redesign, implementation, and results of a core computer science laboratory course (CS376: Linux and Unix Tools). While the course traditionally focused on Unix/Linux environments and system tools, the revised curriculum now introduces dedicated AI learning modules and automated grading. All these changes aim to enhance student learning and reduce instructor workload.

### 1.1 The Core Principles of Using AI in Undergraduate Education

Prior to the course redesign, we surveyed the computer science program's industry advisory board revealing a stark divide in AI adoption. Almost half of the members were strictly prohibited from using generative AI at work, while the other half were encouraged to use them for productivity. The primary differentiator was company size—larger employers were more cautious due to concerns over intellectual property risks. Despite this, there is still an increasing number of companies that use generative AI tools to build software which make the inclusion of teaching generative AI tools in the undergraduate curriculum more and more essential [1, 2, 10, 11].

The discussions among the computer science educators in the department echoed the recent reservations about the use of generative models in education, but we also clearly saw the benefits of using these models in the education of students and their use to automate some mechanistic tasks for software engineers [8, 12].

As a result of the advisory board input, faculty discussions, and student input, we formalized the guiding principles summarized in Figure 1 of how the generative AI tools should and should not be used in the undergraduate computer science education. As the four year undergraduate curriculum progresses from teaching fundamentals to advanced concepts to specialization within the major, the faculty should provide guidance to students on how to use the generative AI tools to aid in their education. The Figure 1 shows an inverse relationship between the student's standing and the reliance on the generative AI models to assist in code generation. In the courses that teach computing fundamentals, generative AI tools should be used to reinforce student understanding of topics by generating quizzes, explaining code, giving an alternative explanation of covered topics, or generating practice problems with solutions. For upper division courses, use of generative AI tools should be focused on more creative tasks such as generating unit tests, bug bashing and code audits, pre-generating code, code commenting and using the generative AI tools as a language interface. The overall guiding principle is to use the generative AI tools as a sounding board or as a tutor just as if it were a field expert. However, students must ensure to avoid over-relying on the tool to do the thinking, writing, coding or problem solving for the student. Off-sourcing the learning, thinking, writing, coding, or problem solving could hinder a program's learning objectives and consequently the student themselves.

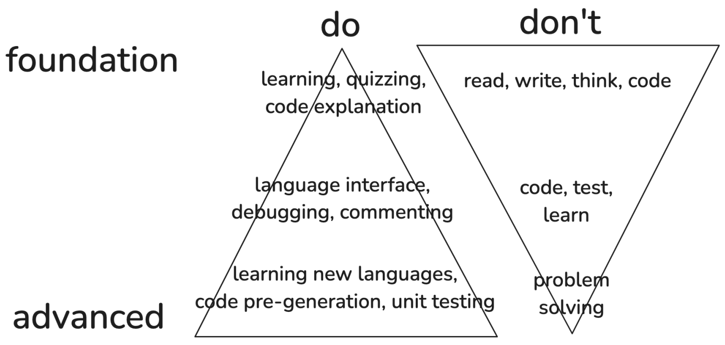
do
don't
foundation
learning, quizzing, code explanation
read, write, think, code
language interface, debugging, commenting
code, test, learn
learning new languages, code pre-generation, unit testing
problem solving
advanced

**Fig. 1.** The guiding principles of how to use AI in the curriculum.

## 2 Curriculum and AI Module Design

The Linux and Unix Tools laboratory, a required one-credit course for second- or third-year students, aims to equip them with practical DevOps skills essential for the workplace [11]. The course covers a broad range of topics, including software development toolchains, build and deployment tools, containerization, virtualization, cybersecurity, and Bash scripting. With generative AI tools becoming an integral part of modern software engineering, their inclusion in the curriculum addresses the growing demand for AI-driven development workflows [2].

The course follows a scaffolded learning approach, where experiential lab modules build on one another through progressive reinforcement of key concepts, ensuring students develop both foundational and advanced skills systematically.

## 2.1 AI Introduction

An increasing number of undergraduate computer science students enter the program with little to no prior exposure to computing beyond basic consumer-level interactions. This growing gap also extends to other aspects of computer science like AI literacy. The introductory section of the AI modules begins with data modeling fundamentals by emphasizing how models generalize patterns from data to approximate user queries. The section starts with a linear regression example, then transitions to word embeddings in high-dimensional vector spaces and retrieval-augmented generation (RAG) in language models.

The basics of prompt engineering are introduced to maintain a balance between providing structured instructions and allowing AI models to generate appropriate responses. Rather than delving into the technical construction of generative AI models, the focus remains on crafting effective prompts to maximize output quality.

Finally, students engage in hands-on exercises exploring natural language processing (NLP) tasks where LLMs excel, such as text summarization, text editing, and document formatting. These structured tasks are later compared to areas where generative AI struggles, including data analytics and precision-based artifact generation which help students understand both the strengths and limitations of AI tools.

## 2.2 AI Fundamentals

Rather than explaining the basics of how generative AI tools are constructed, we treated them as black-box systems focusing on their high-level aspects. Students would explore how the presence or absence of specific training data influenced a model's ability to generate accurate responses. This approach highlighted both the advantages of using generative AI tools, but also emphasized the fundamental limitations of different models to generate correct responses.

To demonstrate the strengths of these models, students worked on tasks such as generating source code for common programming functions, including computing a factorial, sorting a list, and translating text between languages such as English, French, German, and Spanish.

On another task, students were encouraged to explore model failures by selecting a generative AI tool and designing tasks where it would likely struggle. They brainstormed queries for uncommon artifacts absent from the training data, tested highly specific prompt requests, or played a game of 'telephone' to observe how errors compounded over successive generations of model output. Acknowledging that these experiments might yield varying results depending on model updates and retraining, we framed this exercise as an evolving exploration rather than a static test.

Following a think-pair-share pedagogy, students first designed and tested their challenges individually, then collaborated in pairs to critique and refine their examples. Finally, they shared their most interesting model failures on a digital pin-up board, which fostered a broader discussion on bias, generalization, and the unpredictability of AI-generated content.

### 2.3 AI in Discipline

The second experiential learning AI module focused on applying generative AI tools in computer science and software development. This module explored both the benefits and limitations of AI-driven automation in coding, highlighting when AI can enhance productivity and software quality and when it fails to generate reliable results. The goal was to help students identify software engineering tasks that AI can assist with effectively while also recognizing scenarios where AI-generated outputs may be flawed or impractical [2].

The first task was to use LLM for **code explanation** of a non-trivial codebase. Using progressively refined prompts, students worked to generate high-level system descriptions, identify key functional components, and analyze low-level code implementation details. Their comprehension was assessed through small challenge problems that required them to apply their understanding to modified versions of the code. This activity reinforced prompt engineering strategies for extracting meaningful explanations from AI tools.

The second learning objective was to use generative AI tools as a **language interface** between the software project's Specification and Requirement Document (S&R) and its codebase. For this task, students used LLMs to identify features described in the S&R that were missing in the implementation. Similarly, they would need to detect functionalities in the codebase not captured in the documentation. By refining their AI prompts, students gained insight into how LLMs can bridge the gap between technical specifications and implementation while also recognizing the limitations of AI-driven code analysis.

Another focus area is AI-assisted **code commenting** which is a feature that is increasingly integrated into modern integrated development environments (IDE) [2]. Students examined how AI-generated comments corrected, enhanced, or occasionally over-explained existing documentation. These exercises emphasized best practices in comment clarity, scope, and granularity, covering both inline and block-level comments.

The final set of activities explored AI-assisted software engineering tasks such as code-formatting, optimization, and interface design. These were presented as high-level discussions rather than hands-on exercises, as AI-generated output in these domains is highly dependent on the chosen programming language, project-specific coding conventions, and established design patterns [4].

To counterbalance the software engineering tasks for which the use of the generative AI tools improves the software product quality, we asked students to explore one of the tasks on which the generative models fail to produce correct answers. Their reflections included:

– AI-generated code is often syntactically correct but functionally naive, sometimes including unnecessary or inefficient implementations.
– The effectiveness of AI-driven code translation depends on the original code's complexity and the availability of training data for the target language. Translations ranged from poorly structured to usable and inconsistent solutions.
– The outputs of the generative AI tools to optimization code was limited to suggestions and ideas rather then well engineered codebase optimization, especially for the projects implemented in an uncommon programming language.

So far, both AI modules used publicly facing, commercial quality LLMs. The last set of module activities showed students how to interface with a local instance of a pre-trained LLM. The goal of these activities was to illustrate how software engineers can take advantage of **embedding a generative AI model into a software project** with minimal effort and maximum data security. The tasks asked students to step through and analyze each step of processing user prompts from a query written in plain English, transformer conversions into tokens, conversion of the tokens into indexes, embedding the input indexes into the high dimensional vector spaces and generating the model's answers with subsequent output decoding.

### 2.4 AI Ethics

The goal of the AI Ethics exercises was to illustrate how easy it was to infringe on copyright restrictions by using generative AI tool code outputs in a proprietary, for-profit codebase. Building on the AI Fundamentals module, these exercises shifted focus from how AI generates content to the ethical risks of its unchecked use.

Despite efforts by AI developers to train large language models (LLMs) using data covered under fair use doctrine, users of publicly accessible models have no visibility into or control over the training data. As a result, LLMs may inadvertently reproduce copyrighted or copyleft licensed code, raising legal and ethical risks. While these models act as proxies for knowledge generation, they lack the ability to differentiate between openly licensed content and proprietary material, making it essential for users to critically evaluate AI-generated outputs before integration into commercial projects.

## 3 Implementing Automated Grading of Lab Assessments

The CS 376: Linux and Unix Tools course at the University of Portland is designed to equip students with practical system administration skills, covering essential Unix/Linux commands, shell scripting, and DevOps fundamentals. The course follows a hands-on, lab-based format, where students write command-line scripts, analyze system behavior, and document their understanding. However, before integrating Jupyter Notebooks and automated grading, the course relied entirely on manual assessments using PDFs and in-class check-offs. Students documented their command outputs, scripts, and explanations in PDF reports, which were then manually reviewed by instructors. Additionally, for certain lab exercises, students demonstrated their work live to an instructor, who would verify correctness in real time before granting credit.

While this approach worked for small class sizes, it became increasingly inefficient and inconsistent as enrollment grew. The reliance on PDF-based submissions introduced challenges in grading consistency, as instructors had to manually cross-reference each response against expected outputs. Additionally, in-class check-offs became a bottleneck, often leading to long queues of students waiting for their work to be verified. This system also lacked immediate feedback meaning students would often wait days until scheduled class time to receive graded feedback on their assignments. The delay hindered their ability to quickly identify mistakes and reinforce learning ultimately reducing the effectiveness of the hands-on curriculum.

Another major limitation of the previous grading system was the inconsistent computing environments. Students worked on a variety of personal machines with different operating systems, shell environments, and software configurations, leading to execution inconsistencies. Commands that worked on one system could behave differently on another due to variations in macOS vs. Linux utilities, WSL compatibility, and shell differences. These discrepancies made grading subjective, as functionally correct responses could differ in format, requiring instructors to manually investigate system-specific issues rather than focusing on conceptual understanding.

To resolve this, we transitioned to JupyterHub, ensuring all students worked in a standardized, pre-configured Unix/Linux environment. This eliminated OS-related discrepancies, allowing commands to execute consistently across all students. Additionally, JupyterHub streamlined assignment submission and execution through nbgrader, automating structured assessments while providing a reproducible grading environment [6].

With execution environments now standardized through JupyterHub, grading structured components such as code execution results and multiple-choice questions became more efficient with nbgrader. However, free-response answers where students explain Unix/Linux concepts in their own words remained a challenge. These responses varied significantly in phrasing, sentence structure, and level of detail, making rule-based grading methods ineffective.

Traditional keyword-matching approaches often failed to capture semantic correctness, leading to incorrect penalization of valid responses or acceptance of superficially correct but conceptually flawed answers. To address this, we integrated a small language model (SLM) to assess semantic similarity between student responses and expected answers. By using cosine similarity scoring, this approach allows for meaning-based evaluation, ensuring that responses with different wording but correct conceptual understanding are graded fairly.

### 3.1 Grading with SBERT and Semantic Similarity

Evaluating free-response answers in the course posed a unique challenge, as students often expressed correct concepts in varied ways. Traditional rule-based grading systems, such as exact keyword matching or regex-based pattern recognition, struggled to accommodate these variations, leading to false negatives when correct answers were phrased differently and false positives when incorrect responses contained relevant keywords. To overcome these limitations, we integrated Sentence-BERT (SBERT), a transformer-based model designed for semantic similarity tasks, into the grading pipeline [9].

Unlike large language models (LLMs), which require extensive computational resources and often generate responses based on probabilistic text prediction, SLMs like SBERT are optimized for efficient, domain-specific tasks such as similarity comparisons. This makes them well-suited for grading, as they do not generate new content but instead assess meaning by computing the semantic closeness between student responses and expected answers. The decision to use an SLM over an LLM was driven by the need for lightweight, interpretable, and resource-efficient grading, ensuring that assessments could be performed quickly without the overhead of processing massive amounts of data or biased outcomes [3].

The grading pipeline begins with preprocessing student responses, including tokenization, lowercasing, and removing unnecessary formatting. Next, the student's response and a set of predefined expected answers are converted into SBERT sentence embeddings, which represent their meaning in a high-dimensional vector space. Cosine similarity is then computed between the student's response and the expected answer set, yielding a score between 0 and 1, where higher values indicate stronger semantic similarity.

Using these cosine similarity scores, we established a threshold system: responses above 0.7 were automatically marked correct otherwise they were flagged for instructor review. While this system significantly improved scalability and grading consistency, it also introduced new challenges that required ongoing refinement.

### 3.2 Benefits and Challenges of SBERT

The integration of SBERT-based grading into the course provided several key benefits, particularly in scalability, efficiency, and grading consistency. One of the most immediate advantages was reducing instructor workload by automating the evaluation of free-response answers. Previously, grading these responses required manual comparison against a rubric, which became increasingly unsustainable as enrollment grew. With SBERT, a significant portion of responses could be graded instantly, allowing instructors to focus on reviewing borderline cases and improving instructional materials rather than spending hours on routine grading.

Another major benefit was grading consistency. Manual grading often introduced subjectivity, as different instructors might interpret responses differently, leading to inconsistencies in scoring. By applying semantic similarity scoring, SBERT ensured that all responses were graded against the same standardized criteria, minimizing human bias and improving fairness across all students. This was particularly useful for assessing conceptual understanding, as students often used varied wording to explain the same ideas.

Finally, using an SBERT SLM instead of an LLM made the system efficient and lightweight. Unlike large-scale models that require extensive computational resources, SBERT operates efficiently on standard hardware, making it feasible for use within the JupyterHub and nbgrader infrastructure. This ensured that grading could be performed quickly without introducing long processing delays or requiring costly infrastructure upgrades.

Despite its benefits, integrating SBERT into grading introduced several challenges that required continuous refinement. One of the most significant challenges was developing a high-quality, diverse rubric of expected answers. Unlike structured assessments with clear-cut correct and incorrect answers, free-response grading required capturing a broad range of valid explanations. Without sufficient variations in the answer set, the model sometimes underestimated the correctness of valid but unexpected responses, requiring manual intervention to expand and refine the rubric.

Another challenge was the arbitrary nature of cosine similarity thresholds. While 0.7 was selected as a reasonable cutoff based on empirical testing, there was no universally optimal threshold. Raising the threshold risked false negatives, incorrectly marking correct responses as incorrect, while lowering it risked false positives, allowing vague or incorrect answers to pass. Determining the ideal threshold required continuous monitoring and fine-tuning, particularly as new questions and answer variations emerged.

Finally, handling ambiguous cases and borderline scores required ongoing instructor involvement. While SBERT effectively classified clear-cut correct and incorrect responses, cases falling in the gray zone often required human review. This meant that AI-assisted grading did not fully eliminate the need for manual grading, though it significantly reduced the number of responses requiring instructor attention.

### 3.3 Ethical Considerations in AI Grading

The integration of AI in grading introduces important ethical considerations, particularly concerning fairness, bias, and transparency. While SBERT provides a standardized method for evaluating free-response answers, grading automation inherently risks reinforcing biases present in the training data or rubric. If the predefined expected answers are too rigid or fail to account for diverse phrasing, students with different writing styles or language backgrounds may be unfairly penalized. This necessitates continuous refinement of answer sets and instructor oversight to ensure that the model fairly evaluates responses across different student populations.

Another key concern is the opacity of AI-driven decision-making. Unlike manually graded responses, where students can see instructor comments and justifications, AI-graded assessments rely on similarity scores that may not always be intuitive. If a student receives an incorrect grade due to a low similarity score, understanding why their answer was marked incorrect can be difficult. To address this, we incorporated instructor review for borderline cases and provided students with explanatory feedback, ensuring they understand the grading process and have the opportunity to contest their scores if needed.

Despite these challenges, AI-assisted grading can be an ethical enhancement when used responsibly. By reducing instructor bias, providing consistent evaluations, and allowing for rapid feedback, AI supports more equitable and scalable assessments. However, human oversight remains essential to ensure that students are graded fairly, and that the system evolves to accommodate a diverse range of responses and learning styles.

## 4 Delivery and Assessment

The effectiveness of the redesigned CS 376: Linux and Unix Tools course was evaluated through student feedback, performance improvements, and engagement with the new AI-driven learning modules. Table 1 presents a comparative analysis of student assessments from previous course offerings (Fall 2020-2022) and from the redesigned version introduced in Fall 2024. As a one credit laboratory course with no lectures consisting of interactive hands-on learning modules, the assessment categories 2. Effective Communication and 4. Helpful Learning Environment are not a reflection on the instructor's conduct of the course but rather on the effectiveness of learning modules to effectively convey and deliver new course material. The student course assessment of the newly re-designed course show improvements of 16.4%, 21.4%, 21.4%, and 18.8% in the assessment categories 1-4 respectively.

| Assessment Criteria | Fall 2020 (N=18) | Fall 2022 (N=23) | Fall 2024 (N=24) |
|---|---|---|---|
| 1. Valuable Learning Experience | 4.44 | 4.04 | **4.86 (+16.4%)** |
| 2. Effective Communication | 3.94 | 3.43 | **4.50 (+21.4%)** |
| 3. Methods Effectively Conveyed Content | 4.35 | 3.57 | **4.64 (+21.4%)** |
| 4. Helpful Learning Environment | 4.39 | 3.70 | **4.64 (+18.8%)** |

**Table 1.** Student assessment of efficacy delivering course learning outcomes. The reported values are average scores reported by students on a scale from 1: Did not meet to 5: Fully met. The newly redesigned course assessment was delivered in the Fall 2024 while the reported assessment scores for Fall 2020-2022 are for the original course offering. The percentage improvements reported in the last column are calculated between the assessment scores from the Fall 2022 and the newly redesigned course delivery in the Fall 2024.

Qualitative feedback from semi-structured interviews and free-form text responses, visualized in Figure 2, reinforced the positive impact of the course redesign. Common themes included improved engagement, clearer instructional materials, and stronger integration of modern DevOps tools. Students particularly appreciated the structured guidance on generative AI topics, including when and how to use AI tools responsibly in software development. Additionally, the hands-on, self-guided learning format was cited as a major benefit, allowing students to work at their own pace while still receiving immediate feedback.

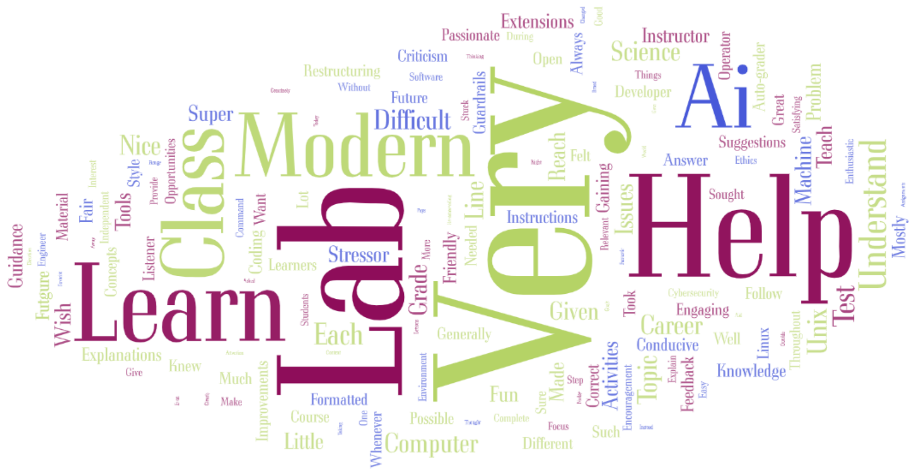

**Fig. 2.** A word-cloud from the students' free-form text responses used by the assessment tool and semi-structured interviews highlight the effectiveness of the newly re-designed course to deliver course content.

Overall, the integration of JupyterHub, nbgrader, and AI-assisted grading significantly improved scalability, grading consistency, and student engagement. While the system required continuous refinement, it demonstrated strong potential as a model for AI-enhanced learning in technical education.

## 5 Conclusion

The increasing accessibility of generative AI tools has introduced both opportunities and challenges in undergraduate computer science education. While these tools have the potential to enhance productivity and learning, they also raise concerns about academic integrity, responsible usage, and long-term implications for software development careers. To address these issues, we implemented a comprehensive course redesign in CS 376: Linux and Unix Tools, integrating experiential learning modules on AI literacy, responsible AI use, and automated grading with small language models.

Our findings suggest several best practices for integrating AI tools in undergraduate education. First, structured guidance on when and how to use AI meaningfully is crucial which we delivered as activities in three thematic categories: AI fundamentals, AI in discipline and AI Ethics. The implementation of the learning activities mainly focused on the use of generative AI tools for productivity and software quality. To counterbalance the desired use of generative AI tools, the modules also included tasks for which LLMs do not produce correct results which highlighted the current limitations of the generative AI models and raised student's awareness for potentially incorrect responses.

Second, the transition to JupyterHub and nbgrader significantly improved the consistency of assessments, particularly for structured programming and multiple-choice tasks. However, grading free-response answers remained a challenge due to the variability in student writing. Implementing SBERT for semantic similarity scoring addressed many of the limitations of rule-based grading by improving flexibility and accuracy, yet it also required extensive fine-tuning of answer sets and threshold calibration to ensure fairness. While this method greatly reduced instructor workload, it did not fully eliminate the need for human oversight, particularly in borderline cases.

Despite the success of automated grading, challenges remain. Developing diverse, high-quality rubric examples is an ongoing effort, as the effectiveness of the model depends on the breadth of accepted answers. Additionally, setting an optimal similarity threshold is inherently subjective and requires continuous refinement. Ethical concerns, such as transparency in grading and potential biases in AI models, further highlight the need for human-in-the-loop oversight to ensure fairness and accountability.

Overall, students responded positively to the course redesign, reporting increased engagement, clearer instruction, and a better understanding of AI's role in software development. The combination of standardized computing environments, automated grading, and structured AI learning modules proved to be an effective approach to modernizing the curriculum while maintaining academic rigor.

**Acknowledgments.** This research project was supported by the Donald P. Shiley School of Engineering Endowment and the Donald P. Shiley School of Engineering Project Development Fund. The authors would like to acknowledge the input from many University of Portland undergraduate computer science students, and the University of Portland Computer Science Advisory Board members whose input was invaluable in the course redesign.

**Disclosure of Interests.** Authors declare no conflict of interest. The course redesign, its implementation, and no aspect of the research to do so were conducted independently without any commercial or financial relationships that would result in a potential conflict of interest.

## References

1. Coutinho, M., Marques, L., Santos, A., Dahia, M., França, C., de Souza Santos, R.: The role of generative ai in software development productivity: A pilot case study.

In: Proceedings of the 1st ACM International Conference on AI-Powered Software. p. 131–138. AIware 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3664646.3664773

2. Ebert, C., Louridas, P.: Generative ai for software practitioners. IEEE Software **40**(4), 30–38 (2023). https://doi.org/10.1109/MS.2023.3265877
3. Flodén, J.: Grading exams using large language models: A comparison between human and ai grading of exams in higher education using chatgpt. British Educational Research Journal **51**(1), 201–224 (2025). https://doi.org/https://doi.org/10.1002/berj.4069
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)
5. Holmes, W., Tuomi, I.: State of the art and practice in ai in education. European Journal of Education **57**(4), 542–570 (2022). https://doi.org/https://doi.org/10.1111/ejed.12533
6. Jupyter, P., Blank, D., Bourgin, D., Brown, A., Bussonnier, M., Frederic, J., Granger, B., Griffiths, T., Hamrick, J., Kelley, K., Pacer, M., Page, L., Pérez, F., Ragan-Kelley, B., Suchow, J., Willing, C.: nbgrader: A tool for creating and grading assignments in the jupyter notebook. Journal of Open Source Education **2**(16), 32 (2019). https://doi.org/10.21105/jose.00032, https://doi.org/10.21105/jose.00032
7. Kitto, K., Sarathy, N., Gromov, A., Liu, M., Musial, K., Buckingham Shum, S.: Towards skills-based curriculum analytics: can we automate the recognition of prior learning? In: Proceedings of the Tenth International Conference on Learning Analytics & Knowledge. p. 171–180. LAK '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3375462.3375526
8. Miao, F., Holmes, W., Huang, R., Zhang, H., et al.: AI and education: A guidance for policymakers. Unesco Publishing (2021)
9. Reimers, N., Gurevych, I.: Sentence-bert: Sentence embeddings using siamese bert-networks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics (11 2019), https://arxiv.org/abs/1908.10084
10. Santos, P.d.O., Figueiredo, A.C., Nuno Moura, P., Diirr, B., Alvim, A.C.F., Santos, R.P.D.: Impacts of the usage of generative artificial intelligence on software development process. In: Proceedings of the 20th Brazilian Symposium on Information Systems. SBSI '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3658271.3658337
11. Sauvola, J., Tarkoma, S., Klemettinen, M., Riekki, J., Doermann, D.: Future of software development with generative ai. Automated Software Engineering **31**(1), 26 (2024)
12. Yu, H.: Reflection on whether chat gpt should be banned by academia from the perspective of education and teaching. Frontiers in Psychology **14** (2023). https://doi.org/10.3389/fpsyg.2023.1181712