

# Generative AI tools to generate summative assessment tasks

Guttorm Sindre<sup>1</sup>[0000-0001-5739-8265], Line Kolås<sup>2</sup>[0000-0003-4908-3241], Robin Isfold Munkvold<sup>2</sup>[0000-0021-2524-279X], Mariusz Nowostawski<sup>1</sup>[0000-0002-2809-8615], Håkon E. S. Ringen<sup>1</sup>, Joakim P. Vassbakk<sup>1</sup>

<sup>1</sup> Department of Computer Science, NTNU

<sup>2</sup> Faculty of Social Sciences, Nord University

**Abstract.** Generative artificial intelligence (GenAI) is increasingly considered for educational purposes, including the design of programming assessments. While prior work has focused on introductory-level courses, little is known about the suitability of GenAI for generating examination tasks at the master’s level. This study investigates the potential of ChatGPT 4.0, Bing Copilot, and Google Gemini, to generate programming exam questions and model solutions. Using a fixed prompt, four researchers collected responses monthly from July 2024 to March 2025. The resulting questions and solutions were analyzed through a mixed-methods approach, combining semantic similarity analysis with Sentence-BERT embeddings, automated code quality metrics, and manual evaluation of task appropriateness. Findings indicate substantial variation across systems: ChatGPT and Gemini generated a diverse set of tasks, including some suitable for advanced assessment, whereas Bing Copilot displayed strong topical convergence, with limited variation and several tasks more appropriate for undergraduate levels. While the vague prompt limited contextual alignment with intended learning outcomes, results show that GenAI can assist educators by providing a pool of candidate tasks and solutions, albeit requiring expert curation and refinement. Future work should examine richer prompting strategies, inclusion of course-specific learning objectives, and student-centered evaluations of generated tasks.

**Keywords:** Programming, Exam Questions, Generative AI, LLMs, Summative Assessments

## 1 Introduction

Generative AI (GenAI) has received a lot of attention among researchers and practitioners in IT education in the last couple of years, especially related to programming courses (Becker et al., 2023). While some of the current debate emphasizes the risks of academic dishonesty (Rytilahti & Kaila, 2024), the implications extend far beyond that and affect both the educational as well as professional domains. For example, software development is increasingly shaped by AI-supported tools, and students must be

prepared for a professional environment in which human expertise is complemented and used alongside the AI systems and tools (Humble et al., 2024). This requires educators to think about new competencies and opportunities that GenAI provides.

One important opportunity is to increase students' and practitioners' competency in prompt engineering (Denny et al., 2023). Another opportunity is to use GenAI to help teaching staff in grading or in assessment of code delivered by students (Gambo et al., 2024). Generative AI may also represent an opportunity for teachers and educational institutions, for instance, to reduce the work-burden or improve the quality of teaching. One example might be to support large-scale teaching and enrich learning resources through AI-based tutoring systems, for example, to extend the reach of instructors in large classes (Ma et al., 2024). Another is to use AI for generating learning resources, assignments and exam questions (Houghty et al., 2024).

The opportunity to use GenAI for assignments and exam questions is the topic of this paper. The main research goal is to understand and compare different generative AI tools, in their ability to generate exam tasks and grading schemes for programming courses. Since most of the research on AI in programming education has focused on introductory courses, we, instead, look at master level courses. Rather than looking at just one AI system, we also want to compare several to get a broader picture of the situation. Moreover, as the capabilities of generative AI tools are developing over time, we also want to investigate the temporal aspect.

Therefore, given the above motivations, we are posing the following research questions:

RQ1: How can GenAI be useful and aid teachers in preparing master level exam questions for programming assessments?

RQ2: What similarities and differences can be found in the generated tasks from the different GenAIs?

RQ3: How stable are LLM-generated task sets when the same prompt and sampling settings are issued monthly over one year? How do they change?

The remainder of this paper is structured as follows: Section 2 reviews related literature on generative AI in education and assessment, with a focus on programming courses. Section 3 describes the research methodology, including data collection and analysis procedures. Section 4 presents the findings from the AI-generated exam tasks and model solutions, with comparisons across tools and over time. Section 5 discusses the implications of these findings for educators and exam design. Section 6 outlines the study's limitations, and Section 7 concludes with recommendations for future research and practical use of GenAI in assessment of programming courses also on master's level.

## 2 Related Work

GenAI can give numerous opportunities within education and learning activities. Bauer et al. (2025) apply the ISAR framework, which categorizes the use of artificial

intelligence in education into four distinct pedagogical approaches: Inversion, Substitution, Augmentation, and Redefinition. Each dimension represents a different way in which AI can be integrated into learning processes, ranging from replacing existing tasks with AI-supported alternatives (Substitution), through enhancing or extending traditional methods (Augmentation), to fundamentally transforming or redefining educational practices (Redefinition). The Inversion category highlights cases where AI shifts the focus of activity or responsibility within the learning environment. Their study reports that the use of AI with the explicit aim of fostering deep learning may, under certain conditions, produce the opposite effect. Instead of supporting higher-order cognitive engagement, some AI applications risk reducing cognitive processing demands, which in turn can negatively affect learning outcomes. This phenomenon exemplifies the inversion dimension of the ISAR framework, where the pedagogical intention is subverted by the actual outcome. Their continued analysis emphasizes that the effectiveness of AI in education is highly contingent upon the knowledge, skills, and motivational readiness of both students and teachers to interact meaningfully with AI systems. Without these prerequisites, the potential benefits of AI integration are unlikely to materialize.

Slimi and Villarejo-Carballido (2024) explore the integration of artificial intelligence in higher education by eliciting expert perspectives on its administrative, pedagogical, and ethical implications. Their study, based on in-depth interviews, highlights the potential efficiency gains in administrative tasks, but underscores significant ethical risks. Most notably algorithmic bias, transparency deficits, and the possible displacement of educational staff. The authors advocate for collaborative stakeholder efforts, robust policy frameworks, and enhanced accountability and fairness in AI deployment, particularly in high-stakes functions like admissions or assessment.

At the same time, using GenAI as a substitute for non-AI instructional support can be used as a viable alternative, see for example Li (2025). According to their meta analysis, GenAI can match traditional instruction on knowledge outcomes while potentially outperforming in applied skills and user engagement.

Logacheva et al. (2024) evaluate the feasibility and reception of contextually personalized programming exercises automatically generated by GPT-4 in an elective introductory (classroom) programming course. Their user study, evaluating output both qualitatively and via student feedback, found that these AI-generated tasks were judged to be of high quality and were engaging and useful to learners. The results suggest that generative AI can provide CS1 instructors with a scalable, personalized pool of practice materials tailored to students' interests and context.

Studies about AI-generated programming tasks vary from the generation of multiple-choice questions (MCQs) to AI-generated code to correct. Doughty et al (2004) explored the application of large language models (LLMs) to create MCQs for programming courses in higher education. Their evaluation included 651 GPT-4-generated and 449 human-authored MCQs, covering 246 learning objectives across six Python courses. Results indicate that GPT-4 can produce MCQs with clear phrasing, a single correct answer, and high-quality distractors. Moreover, the generated questions demonstrated strong alignment with the intended learning objectives. Kohen-Vacs (2025) explored how students interact with AI-generated code during exams and found that the

students faced significantly more difficulty correcting code generated by LLMs than traditional exam tasks, underscoring a distinct challenge AI-generated content introduces in assessment settings.

There have been several papers comparing different generative AI tools, though not for the exact type of task that we gave (suggesting exam questions). Palla and Slaby (2025) compared ChatGPT, Gemini, and Claude, using several different versions of each tool on 10 different programming tasks. A main difference with our prompt was that the tasks were included in the prompt, so that the AI was supposed to solve them, not generate the tasks themselves. The key performance metrics compared between tools were accuracy, reliability, response time, cost, and time efficiency, and Claude 3.5 and 3.0 had the strongest overall performance of the compared tools.

Qiu and Liu (2025) compared several versions of ChatGPT both for answering and generating exam questions. However, this was in the field of medicine, thus less related to programming tasks, though AI was found capable of both tasks.

### 3 Research methods

At the outset of the research, the four involved researchers agreed on the exact prompt to use towards the AI systems. The following prompt was formulated:

*What would be a good programming question for Master-level students studying programming and computer science? Provide the programming question and code that would be the A-level answer, and code that would be E-level answer for this question.*

We also chose which AI systems would be used in the study, namely ChatGPT v 4.0, MS Bing Copilot, and Google Gemini. Of course, even more systems might have been included, like e.g. GitHub Copilot, Claude, DeepSeek. However, too many systems would have made the task prohibitively time-consuming, and three systems were therefore considered to be a suitable trade-off. ChatGPT is a natural inclusion since it is the most well-known and much used both by students and teachers. Bing Copilot was a natural choice because it is the system which the involved universities have a data processing agreement with (as part of the general MS tool package), thus being recommended for teachers and students. Gemini was considered a useful third alternative due to Google's significant market power and possible embedding in its search engine.

#### 3.1 Data collection

The four researchers used the exact same prompt, but otherwise performed the task independently, each giving the prompt to the three chosen systems on the first Monday of every month, starting July 2024 and ending March 2025. The responses were stored in a Teams file archive, with names labelled to be able to separate the user, system and month.

### 3.2 Data analysis

Pairwise semantic distance between responses was investigated automatically using a Python script implementing a lightweight Sentence-BERT model (Reimers and Gurevych, 2019). Further analysis of a subset of the responses was done by two research assistants, partly investigating the responses manually through content analysis, and partly using automated metrics, e.g. for code length, style, and complexity. The analysis combined automated script-based calculations of quantitative metrics (including code complexity measures, semantic similarity scores, and quality indicators) with qualitative evaluation of task appropriateness and content quality. To ensure consistency and reliability, the assistants established shared interpretation criteria for each evaluation dimension and documented their assessments in a structured spreadsheet format. While acknowledging that subjective judgments regarding code complexity, task difficulty, and appropriateness for master's level students may vary between evaluators, efforts were made to reach consensus on ambiguous cases through discussion. The assistants also recorded notable observations, such as instances where AI models exhibited repetitive behavior or generated identical responses across different time periods. This dual approach enabled both quantitative comparison of model performance over time and qualitative assessment of content suitability for advanced programming education.

The metrics used for quantitative analysis of the model solutions were: LOC – lines of code (excluding comments and blank lines in the .py files); Pylint – the quality score of the code given by Pylint; PEP8 – the number of violations against the PEP8 guidelines, according to flake8; CC – the cyclomatic complexity (McCabe, 1976); HV and HD – the Halstead Volume and Halstead Difficulty (Halstead, 1979), respectively; MI the maintainability index (Oman & Hagemester, 1992); and finally, SR – the automatically estimated similarity ratio (0-1) between the A and E model solution by means of Abstract Syntax Tree (Baxter et al., 1998).

## 4 Findings

Many different tasks were generated by the AI systems. Table 1 shows a summary of the topics involved in the selected subset of exam questions that were manually analyzed by the research assistants, sorted according to the highest number of occurrences.

**Table 1.** Topics of the suggested exam tasks.

ChatGPT	Bing	Gemini
Shortest path, Dijkstra (2) Cache system (2) In-memory file system Thread-safe priority queue Huffman code compress. Parking lot (OO, except.)	Shortest path, Dijkstra (6) Web crawler (3) Conditional sum in list	Min. value in graph (2) Shortest path, Dijkstra (2) Matrix multiplication Social network influence Distributed sensor data stream analysis

An immediate observation from Table 1 is that Bing had the least variation in task topics, largely getting “stuck” on two topics: Shortest path problems (typically using Dijkstra’s algorithm in the solution) was used in 6 tasks, and some kind of web crawler featured in 3 tasks. It should be noted, still, that tasks with the same key topic were still not entirely identical, as there was some variation on a more detailed level. For instance, one of the shortest path tasks was supposed to be real-time and dynamic (i.e., obstacles could appear or disappear any time), others had negative weights in addition to positive ones or a requirement for parallelism to be able to process large networks efficiently.

Many of the tasks would not be appropriate in a master level exam. For instance, Bing’s “Conditional sum in list” – generated in the very first month of the study (July 2024) – asked the student to write a program to calculate the sum of all the even numbers in a list of integers. Such a task might be found in an exam of a first semester programming course – and even there, it would be a question on the simple side. Many of the shortest path tasks might be appropriate in an introductory Algorithms course, which students typically take in the second or third semester of a Bachelor.

Most of the tasks featured generic computer science problems, and only two presented “real-world” case descriptions. One of the ChatGPT tasks asked for an OO program to manage a parking lot, and one of the Gemini tasks asked for a program to analyze user influence in a social network / social media application. The prompt did not ask for any real-world case, but it is no surprise that some generated tasks would have this, as some exam tasks in the training material of the systems would likely also feature such case descriptions.

#### 4.1 Model solutions

In addition to proposing exam tasks, the prompt also asked for example solutions – one that would deserve grade A (excellent), and one that would deserve grade E (barely passable). Table 2 shows the results of some automated comparisons of these solutions, across systems and grades, using the metrics explained in section 3.2.

Table 2: Mean values for metrics of the generated model solutions

	LOC	Pylint	PEP8	CC	HV	HD	MI	SR
ChatGPT – A	42.2	7.9	9.0	13.8	82.8	2.5	71.6	0.27
ChatGPT – E	24.4	7.7	4.9	7.9	43.3	1.5	77.9	
Bing Copilot – A	30.9	8.2	5.3	11.2	38.5	1.9	67.7	0.29
Bing Copilot – E	19.6	7.4	4.8	7.7	58.6	1.9	69.8	
Gemini – A	37.3	6.4	8.7	12.6	117.4	2.5	75.6	0.30
Gemini – E	19.3	5.7	4.4	6.4	42.3	1.6	80.6	

It would be too long-winding to go through all these numbers in detail, but there are some notable differences to be seen, both between the various AI systems, and between the model solutions for grades A and E. Generally, E-solutions are shorter and simpler than A-solutions (lower LOC, CC, HV, HD) – with one strange exception: the model solutions from Bing had higher Halstead volume for the E-solution. One possible

reason for this could be if the E-solutions implemented algorithms such as Dijkstra (as was often the case for Bing’s programming tasks) by less practical solutions with excessively intricate code. This could be one way for Bing to distinguish E-solutions from A-solutions. Besides this, a lower Halstead volume is understandable, as the E solution will typically lack some advanced features of the A solution, thus failing to meet all the requirements and constraints. This could lead to a smaller implementation size, yielding a lower HV score. Being shorter, the E-solutions typically also have fewer violations of PEP8 guidelines. Or put another way, the AI systems have not used poor style to distinguish E-solutions from A-solutions.

Among systems, Bing typically has shorter and simpler code than the others, especially considering the A-solutions which are supposed to be the perfect ones. This reflects the finding about tasks given, where Bing had a bigger tendency to get locked into standard algorithms problems like shortest path, which often do not require long code.

For all systems, the average similarity between A- and E-solutions was approximately 0.3. However, there was a wide variation in these numbers. In a few cases, the similarity between A- and E-solutions was as high as 0.5-0.8, typically where both had basically the same approach just that the E-solution was missing some key details. In other cases, the similarity was as low as 0.1-0.2, typically when totally different solution approaches were taken.

#### 4.2 Changes over time, between users and tools

The mean semantic similarity across all pairwise comparisons of AI responses was 0.35, where 1.0 would mean that the content was semantically identical, while 0.0 would mean no semantic overlap at all. The full matrix of pairwise comparisons of semantic similarities is too big to be meaningfully shown and explained. Instead, we show some mean results in Table 3.

Table 3: Some averaged results for pairwise semantic similarities

Comparison	All	Same User	Same Month	User & Month
ChatGPT vs ChatGPT	0.30	0.31	0.31	
Bing vs Bing	0.63	0.62	0.67	
Gemini vs Gemini	0.38	0.35	0.75	
ChatGPT vs Bing	0.32			0.33
ChatGPT vs Gemini	0.27			0.24
Bing vs Gemini	0.37			0.37

The most straightforward observation is that Bing Copilot (0.63, grey) has much higher semantic similarity for same-tool pairwise comparisons of responses than ChatGPT and Google Gemini – or put in another way: Bing has much less variation in its responses. Otherwise, the “All” column indicates that it does not have much effect whether comparisons are same-tool or different-tool, e.g. ChatGPT vs. itself at 0.30, vs. Bing at 0.32, vs. Gemini at 0.27, are all just a few percentage points away from each other. The “Same Person” column shows numbers very similar to the “All” column,

indicating that there is no significant effect on semantic similarity whether prompts are made by the same user or by different users. The “Same Month” column has ChatGPT and Bing with numbers resembling the “All” column, however, Gemini (0.75, grey) has much higher average semantic similarity among responses given in the same month than between responses in general. Finally, the “User & Month” column indicates that for comparisons between different systems, there is no significant effect of user or month, since the numbers here resemble those in the “All” column.

## 5 Discussion

Some of the generated exam questions are not particularly good, either being unsuitable as exam questions, or more appropriate for bachelor level than for master. However, it might be too harsh to portray this as a weakness of the generative AI systems. Notably, the prompt given is rather vague and gives little context. For instance, the prompt does not say anything about the course curriculum or specific learning goals to be tested (except that it is programming for master level CS students). Also, it says nothing about the wanted size and duration of the exam task, nor about the conditions under which the students will be solving it (e.g., what aids they will have available). With such a vague prompt, it might not be surprising that the AI systems end up proposing rather generic programming tasks, some of which seem more appropriate for bachelor than master level.

Another limitation is that only a single prompt is tried, while many teachers might have used a dialogue of several prompts to get good proposals for exam tasks from generative AI. Using more elaborate dialogues would however have been more problematic in a study where one of the key ideas was to compare responses over time and across systems, as the natural follow-up questions in a dialogue would vary between interactions.

Despite the low appropriateness of some tasks, it can also be considered that the AI systems did reasonably well, given the limited amount of information in the prompt. Many of the generated tasks could make sense in a university-level programming course. The generally low scores for semantic similarity (typically around 0.3) also indicates that there is quite a lot of variation between different responses to the same prompt, so that different candidate tasks could be obtained simply by asking repeatedly. The exception to this is Bing Copilot (cf. the grey 0.63 mean similarity in Table 3), which seems to have a much bigger tendency than the others to be locked-in to one topic – specifically related to Dijkstra’s algorithm. The limited range thus revealed by Bing Copilot is somewhat disappointing, especially as being the system which NTNU recommends for GDPR reasons.

Apart from the capabilities of AI, another question that might be posed more from a practitioner’s point of view is to what extent should a teacher outsource the development of exam questions to some AI system? Obviously, even if using AI, the teacher should make a personal expert judgment whether the generated exam question is appropriate, given the intended learning outcomes of the course, the level of the students,

the constraints of the exam situation, etc. In some cases, then, a generated question might be usable as-is, in other cases, only with some modifications, and in yet other cases, not at all. If the teacher already has a good idea for an exam question, it might be better that the teacher develops this manually – or just with some marginal help from AI to flesh out some details – rather than the sort of “blank page” prompts investigated here. At the same time, it is a well-known phenomenon that teachers sometimes struggle to come up with good ideas for new exam questions – especially if the course has already run for some years. If the teacher then ends up making exam questions which are too similar to previous ones (of their own course, or similar courses at other universities), one might consider that getting some new ideas with the help of AI would not have been bad. The other side of that coin, of course, is that the suggestions from AI would not really be new either but based on some exam tasks from other universities seen in some training materials – though they might be “new” in the more localized context of the particular course that the teacher is teaching.

## 6 Limitations

A key limitation of the study is that it just went on for some few months, and with rather few users (4). Since the capabilities of Generative AI are changing fast and their quality is increasing, the results might be somewhat dated already as this is published and even more dated a year or two ahead. Moreover, only 3 generative AI tools were investigated and there are many other versions and tools that would have been equally interesting to explore, such as DeepSeek, Claude, GitHub Copilot. Furthermore, as mentioned above, only a single, relatively simple prompt interaction was tried while most educators would have preferred a dialogue where they could make additional requests to fine-tune the response to the kind of exam task they actually needed.

Another limitation is that the appropriateness of the exam questions was only evaluated analytically, and some of the considerations may be subjective. Alternatively, we might have had a more empirical approach, gathering some CS master students to try to solve the tasks, which could give better insights into some aspects of task appropriateness, e.g., Are the tasks precisely described so that students can solve them? Do the students feel able to demonstrate master-level competencies with these tasks? It should also be noted that the more time-consuming manual analysis was only done for a subset of the responses, so results might have been different if including all responses. However, the automated analysis of semantic differences of the tasks, and of code metrics of the generated solutions, included all the responses, and there is no indication that there was anything special about the manually analyzed tasks versus the rest. Hence, they should be fairly representative.

The study has limited reproducibility. Even if somebody else were to repeat the same prompt with the same versions of the AI tool software after the publication of this paper, they would likely not get the same responses, since the tools might meanwhile have evolved due to increase of the training material. Another issue here is of course the unpredictability of AI responses and lack of transparency concerning how they are

achieved – as even two different persons giving the exact same prompt at the exact same time, might not receive identical answers.

## 7 Conclusions

The main take-away from this investigation is that generative AI systems like ChatGPT and Google Gemini may be usable to aid teachers to generate exam questions with model solutions for rather advanced programming topics. Bing Copilot might also be usable, however, with the rather vague prompt used in this study, it ended up proposing rather simple and limited questions that would be more appropriate in the early stages of a bachelor's degree.

As the results show, the generated tasks were rather generic, and it is likely that the output could have been improved by providing more context in the prompts, such as a list of programming concepts included in the curriculum of the course and a list of learning outcomes to be assessed in the exam, as well as some information on the wanted duration of the task (e.g., half hour, one hour, several hours) and whether the task would make up the entire exam or just be one of several questions in the exam. A natural follow-up to this study would therefore be to make a similar study with more precise prompts, including more context information. Possibly, it might be interesting to ask generative AI to generate an entire exam set consisting of several questions of varying difficulty, and there the totality of questions should cover the learning outcomes of the course in a representative way. Anyway, while GenAI may be a helpful tool for teachers, expert judgment is still necessary for quality assurance of exam tasks and exam sets.

### Acknowledgement

This research work was performed at the Excited Centre for Excellent IT Education, funded by HK-dir under the SFU scheme.

## References

1. Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998, November). Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (pp. 368-377). IEEE.
2. Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023, March). Programming is hard – or at least it used to be: Educational opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 500-506).
3. Becker, B. A., Craig, M., Denny, P., Keuning, H., Kiesler, N., Leinonen, J., ... & Quille, K. (2023). Generative AI in introductory programming. *Computer Science Curricula*, 438-439.
4. Denny, P., Kumar, V., & Giacaman, N. (2023, March). Conversing with copilot: Exploring prompt engineering for solving CS1 problems using natural language. In *Proceedings of the 54th ACM technical symposium on computer science education V. 1* (pp. 1136-1142).

5. Doughty, J., Wan, Z., Bompelli, A., Qayum, J., Wang, T., Zhang, J., ... & Sakr, M. (2024, January). A comparative study of AI-generated (GPT-4) and human-crafted MCQs in programming education. In *Proceedings of the 26th Australasian Computing Education Conference* (pp. 114-123).
6. Gambo, I., Abegunde, F. J., Gambo, O., Ogundokun, R. O., Babatunde, A. N., & Lee, C. C. (2024). GRAD-AI: An automated grading tool for code assessment and feedback in programming course. *Education and Information Technologies*, 1-41.
7. Halstead, M. H. (1979). Advances in software science. In *Advances in Computers* (Vol. 18, pp. 119-172). Elsevier.
8. Humble, N., Boustedt, J., Holmgren, H., Milutinovic, G., Seipel, S., & Östberg, A. S. (2024). Cheaters or AI-enhanced learners: Consequences of ChatGPT for programming education. *Electronic journal of e-Learning*, 22(2), 16-29.
9. Kohen-Vacs, D., Usher, M. & Jansen, M. (2025). Integrating Generative AI into Programming Education: Student Perceptions and the Challenge of Correcting AI Errors. *International Journal of Artificial Intelligence in Education*. <https://doi.org/10.1007/s40593-025-00496-4>
10. Li, J., Yin, K., Wang, Y. et al. Effectiveness of generative artificial intelligence-based teaching versus traditional teaching methods in medical education: a meta-analysis of randomized controlled trials. *BMC Med Educ* 25, 1175 (2025). <https://doi.org/10.1186/s12909-025-07750-2>
11. Logacheva, E., Hellas, A., Prather, J., Sarsa, S., & Leinonen, J. (2024, August). Evaluating contextually personalized programming exercises created with generative AI. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1* (pp. 95-113).
12. Ma, I., Krone-Martins, A., & Videira Lopes, C. (2024, December). Integrating ai tutors in a programming course. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1* (pp. 130-136).
13. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308-320.
14. Oman, P., & Hagemeister, J. (1992, January). Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992* (pp. 337-338). IEEE Computer Society.
15. D. Palla and A. Slaby, "Evaluation of Generative AI Models in Python Code Generation: A Comparative Study," in *IEEE Access*, vol. 13, pp. 65334-65347, 2025, doi: 10.1109/ACCESS.2025.3560244.
16. Qiu, Y. and C. Liu (2025). "Capable exam-taker and question-generator: the dual role of generative AI in medical education assessment." *Global Medical Education*(0).
17. Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*.
18. Ryttilahti, J., & Kaila, E. (2024). How Easy is it to Cheat? – Solving Programming Exercises Automatically with AI. In *Proceedings of the 20th International CDIO Conference, Tunis, Tunisia*.
19. Slimi, Z., & Villarejo-Carballido, B. (2024). Unveiling the potential: Experts' perspectives on artificial intelligence integration in higher education. *European Journal of Educational Research*, 13(4), 1477-1492. <https://doi.org/10.12973/eu-jer.13.4.1477>.

## Appendix : Example GenAI Responses

Here we show one example response from each system. For the sake of uniformity and readability, the code is shown as VSCode screenshots rather than as the raw text stored in the file archive. The Bing response exemplifies that system's preference for questions related to Dijkstra's algorithm. The Gemini response had by far the shortest model answer code, relying heavily on platforms and imported libraries, both for the A-level solution in Java and the E-level solution in Python.

### Example 1: Response from Bing, 5 Nov 2024

A good programming question for Master-level students studying programming and computer science should challenge their understanding of algorithms, data structures, and problem-solving skills. Here's an example question:

**Programming Question:** Implement a function to find the shortest path in a weighted graph using Dijkstra's algorithm. The function should take a graph represented as an adjacency list and a starting node, and return the shortest path to all other nodes.

#### A-level Answer:

```

1  import heapq
2
3  def dijkstra(graph, start):
4      # Priority queue to store (distance, node)
5      pq = [(0, start)]
6      # Dictionary to store the shortest distance to each node
7      distances = {node: float('inf') for node in graph}
8      distances[start] = 0
9      # Set to keep track of visited nodes
10     visited = set()
11
12     while pq:
13         current_distance, current_node = heapq.heappop(pq)
14
15         if current_node in visited:
16             continue
17
18         visited.add(current_node)
19
20         for neighbor, weight in graph[current_node]:
21             distance = current_distance + weight
22
23             if distance < distances[neighbor]:
24                 distances[neighbor] = distance
25                 heapq.heappush(pq, (distance, neighbor))
26
27     return distances
28
29     # Example usage
30     graph = {
31         'A': [('B', 1), ('C', 4)],
32         'B': [('A', 1), ('C', 2), ('D', 5)],
33         'C': [('A', 4), ('B', 2), ('D', 1)],
34         'D': [('B', 5), ('C', 1)]
35     }
36     start_node = 'A'
37     print(dijkstra(graph, start_node))

```

**E-level Answer:**

```

1  def dijkstra(graph, start):
2      distances = {node: float('inf') for node in graph}
3      distances[start] = 0
4      visited = []
5
6      while len(visited) < len(graph):
7          min_node = None
8          for node in graph:
9              if node not in visited:
10                 if min_node is None:
11                     min_node = node
12                 elif distances[node] < distances[min_node]:
13                     min_node = node
14
15                 for neighbor, weight in graph[min_node]:
16                     if distances[min_node] + weight < distances[neighbor]:
17                         distances[neighbor] = distances[min_node] + weight
18
19                 visited.append(min_node)
20
21         return distances
22
23 # Example usage
24 graph = {
25     'A': [('B', 1), ('C', 4)],
26     'B': [('A', 1), ('C', 2), ('D', 5)],
27     'C': [('A', 4), ('B', 2), ('D', 1)],
28     'D': [('B', 5), ('C', 1)]
29 }
30 start_node = 'A'
31 print(dijkstra(graph, start_node))

```

**Explanation:**

The **A-level answer** uses a priority queue (heapq) to efficiently find the shortest path, ensuring optimal performance with a time complexity of  $O((V + E) \log V)$ , where  $(V)$  is the number of vertices and  $(E)$  is the number of edges.

The **E-level answer** uses a less efficient approach, iterating through all nodes to find the minimum distance node, resulting in a higher time complexity of  $O(V^2)$ . This solution is less optimal and slower for larger graphs.

This question tests students' understanding of graph algorithms, efficiency considerations, and their ability to implement complex algorithms correctly.

**Example 2: Response from ChatGPT, 5 Nov 2024**

Here's a question designed to challenge Master-level computer science students, focusing on data structures, algorithms, and efficient problem-solving skills.

---

**Programming Question**

**Title:** Text Compression Using Huffman Coding

**Question:**

Implement a Huffman coding compression algorithm. Given a string, create a compressed binary representation using Huffman coding and provide the corresponding decoding function.

1. **Create a Huffman tree** based on the frequency of characters in the input string.
2. **Generate the Huffman codes** for each character.
3. **Encode the string** into a binary format using the generated codes.
4. **Decode** the binary format back to the original string.

**Requirements:**

- **Input:** A non-empty string of alphanumeric characters.
- **Output:** A tuple `(encoded\_string, codes)` where `encoded\_string` is the binary representation of the original string using Huffman coding, and `codes` is a dictionary containing the mapping of characters to binary codes.
- Implement the **decode function** that takes the encoded string and code dictionary and returns the original string.

**Solution Expectations**

- **A-level answer:** An efficient and well-structured solution that uses appropriate data structures, such as a priority queue, for building the Huffman tree. The code should handle edge cases (e.g., single character strings), demonstrate clear, organized functions, and include comments for readability.
- **E-level answer:** A minimal solution that may produce the correct output but lacks structure, efficiency, or robustness. The code might not handle edge cases and could have redundant steps or unclear logic.

**#### Explanation of A-Level Solution:**

1. **Node Class:** A `Node` class represents each node in the Huffman tree. The `\_\_lt\_\_` method allows nodes to be compared based on frequency, supporting priority queue functionality.
2. **Build Huffman Tree:** The `build\_huffman\_tree` function builds the Huffman tree from character frequencies using a priority queue.
3. **Generate Codes:** `generate\_codes` recursively traverses the Huffman tree to assign binary codes to each character.
4. **Encoding and Decoding:** Encoding and decoding functions efficiently map characters to their binary codes and convert binary back to text.
5. **Efficiency and Structure:** The solution is modular, well-commented, and handles edge cases such as single-character strings.

(A-level code shown on next page)

```
1 import heapq
2 from collections import defaultdict, Counter
3
4 class Node:
5     def __init__(self, char, freq):
6         self.char = char
7         self.freq = freq
8         self.left = None
9         self.right = None
10
11     def __lt__(self, other):
12         return self.freq < other.freq
13
14 def build_huffman_tree(text):
15     frequency = Counter(text)
16     heap = [Node(char, freq) for char, freq in frequency.items()]
17     heapq.heapify(heap)
18
19     while len(heap) > 1:
20         node1 = heapq.heappop(heap)
21         node2 = heapq.heappop(heap)
22         merged = Node(None, node1.freq + node2.freq)
23         merged.left = node1
24         merged.right = node2
25         heapq.heappush(heap, merged)
26
27     return heap[0]
28
29 def generate_codes(node, prefix="", code_dict={}):
30     if node is not None:
31         if node.char is not None:
32             code_dict[node.char] = prefix
33             generate_codes(node.left, prefix + "0", code_dict)
34             generate_codes(node.right, prefix + "1", code_dict)
35     return code_dict
36
37 def encode(text, codes):
38     return ''.join(codes[char] for char in text)
39
40 def decode(encoded_text, codes):
41     reverse_codes = {v: k for k, v in codes.items()}
42     decoded_text = []
43     current_code = ""
44     for bit in encoded_text:
45         current_code += bit
46         if current_code in reverse_codes:
47             decoded_text.append(reverse_codes[current_code])
48             current_code = ""
49     return ''.join(decoded_text)
50
51 def huffman_compress(text):
52     root = build_huffman_tree(text)
53     codes = generate_codes(root)
54     encoded_text = encode(text, codes)
55     return encoded_text, codes
56
57 # Example usage
58 text = "huffman compression"
59 encoded_text, codes = huffman_compress(text)
60 print("Encoded Text:", encoded_text)
61 print("Codes:", codes)
62 print("Decoded Text:", decode(encoded_text, codes))
```

(E-level code:)

```

1  def huffman_compress(text):
2      # Step 1: Calculate frequency of each character
3      frequency = {}
4      for char in text:
5          if char in frequency:
6              frequency[char] += 1
7          else:
8              frequency[char] = 1
9
10     # Step 2: Create a sorted list of tuples (frequency, character)
11     freq_list = sorted(frequency.items(), key=lambda item: item[1])
12
13     # Step 3: Assign binary codes (naive approach)
14     codes = {}
15     binary = '0'
16     for char, freq in freq_list:
17         codes[char] = binary
18         binary = bin(int(binary, 2) + 1)[2:]
19
20     # Step 4: Encode the text
21     encoded_text = ''.join(codes[char] for char in text)
22
23     return encoded_text, codes
24
25 def decode(encoded_text, codes):
26     reverse_codes = {v: k for k, v in codes.items()}
27     decoded_text = ""
28     current_code = ""
29
30     for bit in encoded_text:
31         current_code += bit
32         if current_code in reverse_codes:
33             decoded_text += reverse_codes[current_code]
34             current_code = ""
35
36     return decoded_text
37
38 # Example usage
39 text = "huffman compression"
40 encoded_text, codes = huffman_compress(text)
41 print("Encoded Text:", encoded_text)
42 print("Codes:", codes)
43 print("Decoded Text:", decode(encoded_text, codes))

```

**#### Explanation of E-Level Solution:**

1. **Simplified Frequency Calculation:** Frequencies are calculated but not effectively used to build an optimal tree structure.
2. **Naive Code Assignment:** Instead of building a Huffman tree, this solution assigns binary codes incrementally, without considering optimal compression.
3. **Limited Encoding Efficiency:** The binary encoding scheme is simple but doesn't achieve Huffman's goal of minimizing encoded length based on frequency.
4. **No Structure and Modularity:** Functions lack modular design, with unnecessary repetitions and a lack of comments or clear separation of logic.
5. **Edge Cases Ignored:** It doesn't handle cases like single characters, potentially leading to incorrect decoding.

---

### Summary

- **A-level Solution:** Implements an efficient, structured Huffman coding approach with modular functions and edge case handling.

- **E-level Solution:** A minimal, non-optimal approach that assigns binary codes without building a true Huffman tree, resulting in limited compression efficiency and robustness.

### Example 3: Response from Gemini, 5 Nov 2024

#### Programming Question:

Design and implement a distributed system to efficiently process and analyze large-scale sensor data streams in real-time.

#### A-Level Answer:

##### Key Considerations:

- **Scalability:** The system should handle increasing data volumes and processing demands.
- **Real-time Processing:** Low-latency processing is crucial for timely insights.
- **Fault Tolerance:** The system should be resilient to failures and able to recover quickly.
- **Data Consistency:** Ensure data integrity and consistency across distributed nodes.

##### Proposed Architecture:

- **Data Ingestion Layer:**
- **Kafka:** A distributed streaming platform for real-time data ingestion.
- **Kafka Connect:** For efficient data ingestion from various sources (sensors, databases, etc.).

##### Data Processing Layer:

- **Apache Flink:** A distributed streaming engine for real-time data processing and analysis.
- **Windowing and Aggregation:** Group data into time windows and apply aggregation functions (e.g., average, sum, count).
- **State Management:** Maintain stateful computations for tracking trends and anomalies.

##### Data Storage Layer:

- **Time-Series Database:** A specialized database (e.g., InfluxDB, TimescaleDB) for efficient storage and retrieval of time-series data.

##### Visualization Layer:

- **Dashboards:** Create interactive dashboards using tools like Grafana or custom web applications.

- Real-time Visualization: Update visualizations in real-time to reflect the latest data.

```

1  DataStream<SensorReading> sensorDataStream = env.addSource(new SensorSource());
2
3  // Window and aggregate data
4  DataStream<Tuple2<String, Double>> avgTemp = sensorDataStream
5     .keyBy("sensorId")
6     .timeWindow(Time.seconds(5))
7     .aggregate(new AvgTemperature());
8
9  // Write aggregated data to a time-series database
10 avgTemp.print();
11 avgTemp.addSink(new InfluxDBSink());

```

### E-Level Answer:

#### Key Considerations:

- Basic Data Processing: Focus on simple data cleaning and transformation.
- Single-Node Processing: Process data on a single machine.
- Batch Processing: Process data in batches rather than real-time.

#### Proposed Architecture:

- Data Ingestion: Read data from a file or database.
- Data Processing: Use Python libraries like Pandas and NumPy for data cleaning, filtering, and basic analysis.
- Data Visualization: Use Matplotlib or Seaborn to create static visualizations.

#### Code Example

```

1  import pandas as pd
2  import matplotlib.pyplot as plt
3
4  # Read sensor data from a CSV file
5  df = pd.read_csv("sensor_data.csv")
6
7  # Clean and filter data
8  df_cleaned = df.dropna()
9  df_filtered = df_cleaned[df_cleaned["temperature"] > 30]
10
11 # Calculate average temperature
12 avg_temp = df_filtered["temperature"].mean()
13
14 # Visualize data
15 plt.plot(df_filtered["timestamp"], df_filtered["temperature"])
16 plt.xlabel("Timestamp")
17 plt.ylabel("Temperature")
18 plt.title("Sensor Data")
19 plt.show()

```

**Note:** This is a simplified example. A full-fledged distributed system would require more complex considerations, such as fault tolerance, load balancing, and security.