Certamen Artificialis Intelligentia: Evaluating AI in Solving AI-generated Programming Exercises

Carmine Coppola, Simone Perrotta, Ciro Giuseppe De Vita, Gennaro Mellone, Diana Di Luccio, and Raffaele Montella
University of Naples "Parthenope"
Naples, Italy
{carmine.coppola001,simone.perrotta001, gennaro.mellone1}@studenti.uniparthenope.it
{cirogiuseppe.devita,diana.diluccio, raffaele.montella}@uniparthenope.it

Josè Carlos Paiva and Ricardo Queiros
Polytechnic of Porto
Porto, Portugal
josepaiva@sc.ipp.pt, ricardoqueiros@esmad.ipp.pt

Robertas Damasevicius and Rytis Maskeliunas
Kaunas University of Technology
Kaunas, Lithuania
{robertas.damasevicius,rytis.maskeliunas}@ktu.lt

Jakub Swacha
University of Szczecin
Szczecin, Poland
jakub.swacha@usz.edu.pl

Abstract

Large language models (LLMs) are transforming programming education by enabling automated generation and evaluation of coding exercises. While previous studies have evaluated LLMs' capabilities in one of these tasks, none have explored their effectiveness in solving programming exercises generated by other LLMs. This paper fills that gap by examining how state-of-the-art LLMs—ChatGPT, DeepSeek, Qwen, and Gemini—perform when solving exercises generated by different LLMs. Our study introduces a novel evaluation methodology featuring a structured prompt engineering strategy for generating and executing programming exercises in three widely used programming languages: Python, Java, and JavaScript. The results have both practical and theoretical value. Practically, they help identify which models are more effective at generating and solving exercises produced by LLMs. Theoretically, the study contributes to understanding the role of LLMs as collaborators in creating educational programming content.
Keywords: Large Language Models, Gamified Programming Exercises, AI-driven Assessment, Automated Code Evaluation.

1. Introduction

Large Language Models (LLMs) are increasingly integrated into programming education, offering benefits such as personalization, scalability, and accessibility [1], [7], [9]. These models, trained on vast text and code corpora, can generate exercises aligned with pedagogical goals [4, 5] and solve them [2], making them useful in both instruction and assessment.

However, challenges remain. Training biases [11] and output stochasticity [6] can affect exercise quality and consistency. Despite this, their adoption in education continues to expand.

While prior benchmarks have examined LLMs either as generators [2] or solvers [3], no study has assessed their performance across both roles in a unified setting. This paper addresses that gap by evaluating four widely used LLMs—ChatGPT, DeepSeek, Qwen, and Gemini—in generating and solving programming exercises produced by their peers.

2. Experimental Procedure

2.1. Exercise Generation

Exercises were generated using a structured prompt engineering approach to ensure consistency and pedagogical soundness. Each LLM was prompted to create tasks of varying difficulty based on a predefined prompt. The resulting outputs were verified for correctness and format before being used in the experiment.

The exercise generation process employed the following API versions to ensure experimental consistency: GPT-4 (OpenAI), DeepSeek-R1, Qwen-Turbo, and Gemini 2.0 Flash.

Figure 1 presents the prompt used to generate programming exercises for the experiment.

```
# Prompt for generating the exercises
        prompt = """
        Generate a list of 10 programming exercises in JSON format. Each exercise must strictly follow this structure:

        [
            {
                "name": "<A concise and clear title for the exercise>",
                "description": "<A detailed description of the exercise, specifying exactly what needs to be implemented>",
                "input": [<A list of example inputs of the correct type that will be used to test the solution>],
                "output": [<A list of expected outputs of the correct type corresponding to each input>]
            }
        ]

        ### Constraints:
        1. Each exercise must explicitly define a **set of input values** in the `"input"` field.
        2. Each exercise must explicitly define the **expected output values** in the `"output"` field.
        3. The **number of elements in `"input"` and `"output"` must always match**, ensuring testability.
        4. **Data types must be correct for both inputs and outputs**:
            - If the function expects **integers**, provide integers in `"input"`, not strings.
            - If the function expects **floating-point numbers**, use decimals (e.g., `3.14`).
            - If the function expects **strings**, provide strings.
            - If the function expects **boolean values**, return actual boolean values (`true/false`), not strings.
            - If the function expects **lists**, provide lists with elements of the correct type.
            - If the function expects **dictionaries (maps)**, provide correctly formatted JSON objects.
        5. Ensure diverse exercises covering different programming concepts such as **loops, recursion, data structures, and algorithms**.
        6. The **JSON output must be properly formatted and valid**.
        """
```

Fig. 1. Generator prompt

2.2. Exercise Solving

The Executor prompt was designed as to ensure not only that the LLM will generate a solution in the form of a source code in the indicated programming language but also that its output will adhere to the specified format requirements so that the evaluation process could be streamlined. Figure 2 presents the prompt used to solve programming exercises in the experiment.

```
"""
        Generates a prompt that strictly forces the LLM to output ONLY a valid JSON object
        without any extra text, comments, or markdown.
        """
        if data and "name" in data and "description" in data:
            return f"""
        You are an automatic code generator.

        Task:
        Given the exercise below, return ONLY a valid JSON object in your response.
        Do NOT add any comments, markdown, explanations, or extra text.

        Format (STRICT):
        {{
            "exercise": "{data['name']}",
            "solution": "<COMPLETE CODE AS A SINGLE STRING>"
        }}

        - The solution must be valid {self.language} code for a function named 'solution', as shown in the template below.
        - The function must take as input all required parameters, process them as described, and return ONLY the result (no print, no extra output).
        - If the exercise has more than one argument, ensure the function signature is correct.
        - If the language is Java, include a full class with a static main that reads from args.
        - DO NOT include any explanation, comments, markdown, triple backticks, or any text outside the JSON.

        Exercise: {data['name']}
        Description: {data['description']}

        Template:
        {{
            "exercise": "{data['name']}",
            "solution": {json.dumps(self.generate_function_template())}
        }}
        """
```

Fig. 2. Executor prompt

2.3. Experimental Setup and Metrics

The experiment was run locally to avoid cloud-related inconsistencies. Each LLM acted as both generator and solver in four rotations, generating 20 exercises per round. All models, including

the generator, attempted to solve each set of tasks in Python, JavaScript, and Java, ensuring language diversity and cross-evaluation.
We used three metrics to evaluate performance:

- $Err_s = \frac{N_{syntax}}{N_{ex}}$: the proportion of solutions with syntax errors (e.g., uncompilable or improperly formatted code);
- $Err_l = \frac{N_{logical}}{N_{ex}}$: the proportion of syntactically correct solutions that produce incorrect outputs; these may be due to the executor generating an implementation that does not fulfill the task requirements, or the generator providing an imprecise task definition;
- $TE = \frac{N_{syntax}+N_{logical}}{N_{ex}} \cdot 100\%$: the total error rate, combining both syntax and logic errors.

3. Results

We report the results of our cross-evaluation experiment, in which each LLM attempted to solve exercises generated by itself and the other LLMs.

Table 1 reports the syntax error rates Err_s across generator–executor combinations. Rows represent generators; columns, executors. The **Mean for Generators** column reflects how easily exercises from each model are solved. The **Mean for Executors** row shows which models better solve others' tasks. Lower values in both indicate stronger performance.

Executor→ Generator↓	**ChatGPT**	**DeepSeek**	**Qwen**	**Gemini**	**Mean for Generators**
ChatGPT	0.043	0.044	0.061	0.045	0.048
DeepSeek	0.056	0.053	0.072	0.061	0.061
Qwen	0.072	0.067	0.068	0.056	0.066
Gemini	0.056	0.061	0.065	0.070	0.063
Mean for Executors	0.057	0.056	0.066	0.058	Err_s

Table 1. Comparing Err_s

Key findings of Table 1: *ChatGPT emerges as the generator with the lowest average syntax error (4.8%), while Qwen shows the highest (6.6%). As executors, ChatGPT, DeepSeek, and Gemini perform similarly (around 5.7%), whereas Qwen is the least reliable, again with 6.6%.*

Table 2 presents the results regarding the rate of logical errors Err_l in the generated solutions.

Executor→ Generator↓	**ChatGPT**	**DeepSeek**	**Qwen**	**Gemini**	**Mean for Generators**
ChatGPT	0.005	0.012	0.013	0.011	0.010
DeepSeek	0.006	0.011	0.028	0.006	0.012
Qwen	0.017	0.017	0.016	0.011	0.015
Gemini	0.017	0.022	0.033	0.019	0.023
Mean for Executors	0.011	0.015	0.023	0.012	Err_l

Table 2. Comparing Err_l

Key findings of Table 2: *ChatGPT also proves to be the most reliable generator in terms of logic, with a mean logical error rate of 1.0%, followed by DeepSeek (1.2%), while Gemini reaches the highest (2.3%). As executors, ChatGPT (1.1%) and Gemini (1.2%) perform best, while Qwen again records the worst result (2.3%).*

Table 3 presents the total error rates TE observed across all generator–executor combinations.

Executor→ Generator↓	ChatGPT	DeepSeek	Qwen	Gemini	Mean for Generators
ChatGPT	4.8	5.6	7.4	5.6	5.9
DeepSeek	6.1	6.4	10.0	6.7	7.3
Qwen	8.9	8.3	8.3	6.7	8.0
Gemini	7.2	8.3	9.8	8.9	8.6
Mean for Executors	6.8	7.2	8.9	7.0	$TE(\%)$

Table 3. Comparing the overall total error index (TE).

Key findings of Table 3: *Considering the total error index (TE), ChatGPT confirms its leading role as both generator (5.9%) and executor (6.8%). Gemini results in the least reliable generator (8.6%) and Qwen the least effective executor (8.9%).*

Although all error rates stayed below 10%, their analysis highlights performance variability and helps assess each model's suitability as both generator and solver in educational contexts.

4. Discussion

Recent studies [2, 3], [8], [10] show that LLMs perform well on well-defined programming tasks, but struggle with ambiguous, creative, or complex challenges, and often produce code with syntax or logical errors. Prompt quality plays a key role in performance. Among these studies, ChatGPT was the most effective solver, correctly addressing about 60% of tasks [3].

We tested four LLMs, each generating 20 exercises and attempting to solve exercises generated by others, including their own.

We assessed generation quality based on the average solver error rate per generator. ChatGPT proved the most effective (5.9%), while Gemini yielded the highest error rate (8.6%).

5. Conclusion

Our results confirm that LLMs can both generate and solve programming exercises across multiple programming languages, with ChatGPT notably outperforming the other LLMs.

While our approach is novel in considering the double role of LLMs , it has some limitations and does not explore all research opportunities it opens. Its main limitation is the absence of human evaluation, which could expose generated exercises that may be ambiguous or awkward despite being solvable by AI. Moreover, the study does not include a systematic analysis of potential biases, such as consistent differences in difficulty or implicit preference for specific programming paradigms. These will be addressed in future work.

Acknowledgements

This research was co-funded by the European Union, grant 2023-1-PL01-KA220-HED-000164696. The authors would like to thank Francesco Peluso from University of Naples "Parthenope" for his involvement in the reported research.

References

[1] Cain, W.: Prompting change: exploring prompt engineering in large language model ai and its potential to transform education. TechTrends 68(1), pp. 47–57 (2024)

[2] Hou, W., Ji, Z.: Comparing large language models and human programmers for generating programming code. Advanced Science 12(8), pp. 2412279 (2024)

[3] Huang, Y., Lin, Z., Liu, X., Gong, Y., Lu, S., Lei, F., Liang, Y., Shen, Y., Lin, C., Duan, N., Chen, W.: Competition-level problems are effective llm evaluators (2024), https://arxiv.org/abs/2312.02143

[4] Joel, S., Wu, J.J., Fard, F.H.: A survey on llm-based code generation for low-resource and domain-specific programming languages (2024), https://arxiv.org/abs/2410.03981

[5] Montella, R., De Vita, C.G., Mellone, G., Ciricillo, T., Caramiello, D., Di Luccio, D., Kosta, S., Damaševičius, R., Maskeliūnas, R., Queiros, R., Swacha, J.: Leveraging Large Language Models to Support Authoring Gamified Programming Exercises. Applied Sciences 14(18), pp. 1–15 (2024)

[6] Ouyang, S., Zhang, J.M., Harman, M., Wang, M.: Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. arXiv:2308.02828 (2023)

[7] Stamper, J., Xiao, R., Hou, X.: Enhancing LLM-based feedback: Insights from intelligent tutoring systems and the learning sciences. In: International Conference on Artificial Intelligence in Education. pp. 32–43. Springer (2024)

[8] Svetkin, A.: Testing llms on solving leetcode problems. https://hackernoon.com/testing-llms-on-solving-leetcode-problems (2024), accessed on Feb 18, 2025

[9] Tapalova, O., Zhiyenbayeva, N.: Artificial intelligence in education: Aied for personalised learning pathways. Electronic Journal of e-Learning 20(5), pp. 639–653 (2022)

[10] Walker, S.M.: Bigcodebench: A new benchmark for evaluating LLMs on programming tasks. https://klu.ai/glossary/bigcodebench-eval (2024), accessed on Feb 18, 2025

[11] Warr, M., Oster, N.J., Isaac, R.: Implicit bias in large language models: Experimental proof and implications for education. Journal of Research on Technology in Education pp. 1–24 (2024)