Analysis of LLM Code Generation Through Pylint

Abhishek Manhas, Quoc Do, Nathan Ferrence, Eric Liu, Chris Lee

April 2025

Contents

1 Introduction

Recent advances in Large Language Models (LLMs) have significantly improved the quality of code generation. These models, trained on massive code train dataset, are now capable of solving a wide range of programming problems across various domains. However, the quality of the code produced—beyond functional correctness—remains a critical and underexplored area. In this study, we examine how well code generated by different LLMs adheres to coding best practices by evaluating their output using `pylint`, a widely used static code analyzer.

We selected a diverse set of five programming problems, ranging from sorting algorithms to graph theory and string parsing. For each problem, we prompted several LLMs—including DeepSeek, GPT-4o, Gemini, Claude, and Perplexity—to generate Python solutions. The code produced was not edited and was directly analyzed with `pylint` to assess its code quality based on predefined standards such as naming conventions, code readability, unused imports, and other logical constructs.

Our primary research objectives are twofold:

- **RQ1:** How do various LLMs compare in terms of static code quality as assessed by `pylint`?
- **RQ2:** What specific coding issues are most common among LLM-generated code? (Looking through the documentation)

2 Methodology

We designed a test suite consisting of five diverse programming challenges:

1. Sorting a list of integers.
2. Solving a constrained variant of the Traveling Salesman Problem.
3. Subgraph detection between two undirected graphs.
4. Expression evaluation using a basic calculator.
5. Scheduling CPU tasks with cooldown intervals.

Each problem was posed with a consistent prompt instructing the LLM to write Python code that would be evaluated using `pylint` to emphasize code quality. No modifications were made to the generated code before analysis.

The LLMs tested include:

- DeepSeek V3 and R1[2]
- GPT-4o, GPT-o3, o4-mini high [4]

- Gemini 2.0 Flash, 2.5 Pro, and 2.5 Flash [3]
- Claude 3.7 Sonnet [1]
- Perplexity [5]

Each generated solution was scored using `pylint`, which provides a score from 0 to 10 based on the quality of the code, where higher values indicate better adherence to coding standards.

3 Research Questions Results

3.1 RQ1: How do the models rank in terms of static code quality?

We tabulated the average `pylint` scores for each LLM across all five problems. The results are summarized in Table 1 in the Results section.

Model	Sort	TSP	Subgraph	Calculator	CPU Tasks
DeepSeek V3	8.57	6.92	7.78	9.41	8.26
DeepSeek R1	7.86	7.74	8.92	8.56	5.71
GPT-4o	10	9.6	9.7	9.71	9.67
GPT-o3	10	10	10	10	10
GPT-4 mini High	10	9.25	10	10	10
Gemini 2.0 Flash	8.33	9.39	8.89	9.7	8.33
Gemini 2.5 Pro	9.78	8.81	9.57	9.5	6.43
Gemini 2.5 Flash	2.31	7.14	9.25	9.35	5.56
Claude 3.7 Sonnet	6.67	3.68	5.38	5.0	4.89
Perplexity	8.0	9.52	9.03	9.64	9.23

Table 1: `pylint` scores (out of 10) for each LLM across 5 problems

3.2 RQ2: What criteria does Pylint use?

`pylint` analyzes code based on several factors including:

- Proper naming conventions
- Code layout and formatting
- Presence of unreachable or dead code
- Use of built-in functions
- Avoidance of redundant statements
- Import correctness
- etc...[6]

4 Discussion

4.1 GPT

The GPT series, particularly GPT-4o, GPT-o3, and GPT-4 mini high, exhibited exceptional code quality across all tasks. GPT-o3 and GPT-4 mini high achieved perfect scores of 10/10 on each problem, demonstrating not only functional correctness but a remarkable adherence to Python best practices. GPT-4o closely followed with near-perfect ratings in every category.

The GPT models consistently avoided common linting pitfalls. Most of the code was well-documented, followed proper naming conventions, and maintained logical structure. The very few deductions in score came from minor issues such as, C0325: Unnecessary parentheses after not, C0103: Non-conforming constant names (testexpr, cooldown), R0914: Too many local variables in a single function, R1730: Use of conditional blocks instead of built-in functions like min().

Despite these small warnings, GPT models exhibited an excellent balance between readability, structure, and functionality. Their scores ranged from **9.25 to 10.00**, with the majority of generated code maintaining a **10/10 rating** across runs. This consistency highlights the maturity of OpenAI's models in not just solving problems, but doing so with production-grade code quality in mind.

4.2 Deepseek

The choice of models, **Deepseek-V3** and **Deepseek-R1**, reflects an intent to compare iterations of a generative AI system for code-writing capability. Deepseek-V3 produced generally higher scores, with the Basic Calculator solution receiving the highest rating of **9.41/10**, followed by Sorting Array **(8.57/10)** and Task Scheduling **(8.26/10)**. On the other hand, Deepseek-R1 showed more variance in performance. It performed well in the Graph Subgraph Check **(8.92/10)** but poorly in Task Scheduling, scoring just **5.71/10**—the lowest among all tests.

While the scores are mostly reasonable, they are not always aligned with expectations based on problem complexity. For instance, problems like Sorting Array and Basic Calculator are structurally simple but still had minor issues, such as missing final newlines and insufficient public methods. Meanwhile, more complex problems like the Graph Subgraph Check managed to achieve respectable scores, albeit with warnings about excessive local variables and import style.

Across both models, the most common errors were related to style conventions, including **missing final newlines (C0304)**, **trailing whitespace (C0303)**, variable or module names not following the **snake_case naming**

convention (C0103), and **missing module docstrings (C0114)**.

In conclusion, both Deepseek-V3 and Deepseek-R1 generated mostly functional code, but with common style issues like missing newlines, improper naming, and lack of documentation. Deepseek-V3 showed more consistent quality with higher average scores, while Deepseek-R1 had more variability and lower scores on some problems. The main weakness across both models lies in Python convention violations, not logic errors.

4.3 Gemini

The three Gemini models—**2.0 Flash, 2.5 Pro, and 2.5 Flash**—exhibited notable variation in code quality. While newer models sometimes produced higher-scoring outputs on individual tasks, Gemini 2.0 Flash maintained the most consistent pylint performance across the board.

Gemini 2.0 Flash averaged a Pylint score of **8.93**, with all five problems scoring above 8. The most common issues were stylistic: C0103 for non-snakecase variable names, W0612 for unused variables, and missing docstrings (C0114, C0116). It produced particularly clean code on the calculator problem (9.70) and the TSP solver (9.39), with no logic-related warnings.

Gemini 2.5 Pro had a slightly lower average of **8.82**. It generated strong results for sorting (9.78), subgraph detection (9.57), and calculator (9.50), but significantly underperformed on CPU scheduling (6.43). Issues included W0613 for unused function arguments, R1705 for unnecessary else blocks after returns, and line-length warnings (C0301). These indicate a mild drop in structural discipline, particularly on problems requiring control flow management.

Gemini 2.5 Flash was the least consistent, averaging only **6.72**. Though it matched 2.5 Pro on subgraph (9.25) and calculator (9.35), it performed poorly on sorting (2.31) and CPU scheduling (5.56). Major problems included unreachable code (W0101), undefined variables (E0602), overly complex branching (R0912), and unused imports (W0611). These suggest that code generation in this lighter model sacrifices quality for speed.

Overall, despite updates in later versions, Gemini 2.0 Flash delivered the most consistently clean code. Gemini 2.5 models showed strengths on algorithm-heavy problems but frequently stumbled on stateful or branching logic. This illustrates that model updates and speed improvements do not uniformly translate to better static code quality.

4.4 Claude

The Claude model **did not perform as expected**. It was expected to be one of the higher scoring models. This is because it is designed to output code. When looking at the results, it can be seen to have the most consistently low scores. There was only one instance being problem A, where **Gemini 2.5 flash** had a lower score. The main error Claude was receiving was from **"trailing whitespace"**. Another consistent error received was **"final newline missing"**. For problem D, the model made constant test names where points were lost due to the upper case naming style.

4.4.1 Perplexity

Perplexity had high scores, **being above 9**, except for problem A. Looking at the score for problem A, 8 can be seen from **"missing module docstring"**. This was seen in all the problems, Perplexity did not make any docstrings to start the files. The only other error that was found by Pylint was in problem C. This error was found with the argument name not conforming to the **"snake_case"** naming style. This model performed as expected since it is a combination of many other models for the best performance.

5 Conclusion

While most LLMs produced functionally correct code, their static code quality varied significantly across models. GPT-4 variants, particularly GPT-o3 and GPT-4 mini high, consistently scored at or near the maximum on all problems, demonstrating excellent conformance to Python best practices and style conventions. These models not only solved the problems correctly but also did so with production-ready formatting, structure, and naming, making them stand out in terms of both logic and readability.

In contrast, models like Claude and Deepseek-R1 showed inconsistent results, with common issues including missing docstrings, improper naming conventions, and trailing whitespace. The Gemini models presented a mix—Gemini 2.0 Flash was the most consistent, while 2.5 Flash showed signs of instability and performance tradeoffs. Perplexity generally produced high-quality code, with minor stylistic lapses like missing docstrings, even though its a web scraper. These results underscore the importance of evaluating AI-generated code not just for correctness but also for maintainability and style.

References

[1] Anthropic. Claude ai. https://www.anthropic.com/claude, 2025. Accessed: 2025-05-04.

[2] DeepSeek-AI. Deepseek-v3. https://github.com/deepseek-ai/DeepSeek-V3, 2025. Accessed: 2025-05-04.

[3] Google AI. *Gemini API Models Documentation*, 2025. https://ai.google.dev/gemini-api/docs/models.

[4] OpenAI. *OpenAI API Models Documentation*, 2025. https://platform.openai.com/docs/models.

[5] Perplexity AI Team. Open-sourcing r1 1776, 2025. Accessed: 2025-05-04.

[6] Pylint contributors. *Pylint Documentation*, 2025. https://pylint.readthedocs.io/en/stable/.