Do LLMs support generating code that is compliant with creational design patterns?

Caio S. Machado[1], **Eduardo Kugler Viegas**[2], **Silvana Morita Melo** [3], **Leo Natan Paschoal**[2]

[1] Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo, (USP) – São Carlos – SP – Brasil

[2] Pontifícia Universidade Católica do Paraná, (PUC-PR) – Curitiba – Paraná – Brasil

[3] Universidade Federal da Grande Dourados, (UFGD) – Dourados – Mato Grosso do Sul – Brasil

caiomachado3a@gmail.com, eduardo.kugler@pucpr.br

silvanamelo@ufgd.edu.br, leo.paschoal@pucpr.br

Abstract. *Large Language Models (LLMs) have been increasingly employed in Software Engineering; however, their reliability in applying established practices such as design patterns remains uncertain. This study investigates the ability of the LLMs ChatGPT, Gemini, and Copilot to generate code that adheres to creational design patterns. The analysis considers the correctness of the implementations, as well as the complexity and maintainability of the generated code. To this end, a comparative study was conducted involving 150 programs generated by the models and 50 reference programs developed by humans. The results reveal that, although LLMs are generally capable of applying design patterns, a critical failure rate of 22.67% was observed, with significant performance variation across models and design patterns. Regarding code complexity, no substantial differences were identified; however, in most cases, the artifacts generated by the LLMs exhibited higher maintainability than the human-written reference implementations.*

1. Introduction

Large Language Models (LLMs) are capable of generating various types of content, such as text and source code, from patterns extracted from extensive training datasets [Feuerriegel et al. 2023]. In the context of Software Engineering, these models have been adopted to support development teams in several activities, including coding [Li et al. 2023], automated test generation [Liu et al. 2023], and software documentation [Nguyen-Duc et al. 2023]. Thus, LLMs have established themselves as relevant tools in supporting systems development.

Despite the growing adoption of this technology, Software Engineering is founded on established practices to ensure the quality of the systems produced. Among these practices, *design patterns* stand out, defined as reusable and validated solutions for recurring problems in software development [Gamma et al. 2007]. Such patterns constitute a fundamental mechanism to guide software construction. It is widely recognized that code that follows these guidelines tends to exhibit higher quality, especially in terms of maintainability, reusability, and complexity control [Riehle 2011].

With the popularization of the use of LLMs by professionals in the field for writing code, the quality of the generated artifacts has become a central topic of investigation [Simões and Venson 2024, Jamil et al. 2025]. Recent studies indicate that, although useful, these models can produce code with flaws, inconsistencies, and logical errors, which represent risks to the integrity of software projects [Kabir et al. 2024].

Considering that developers resort to design patterns as a strategy to ensure code quality, and that, in parallel, they use LLMs to accelerate the coding stage, a fundamental question emerges: *Does the code generated by LLMs that purports to follow a design pattern maintain the expected level of quality?* Investigating whether the application of a pattern by an LLM results, in fact, in high-quality code is, therefore, an essential task.

In this context, this work aims to evaluate the ability of the LLMs ChatGPT[1], Gemini[2], and Copilot[3] to generate code that adheres to creational design patterns. To this end, the study analyzes the correctness of each pattern's implementation and evaluates the quality of the generated code through complexity and maintainability metrics.

2. Design patterns

Design patterns are proposed as a mechanism to facilitate the production and ensure software quality. They do not function as ready-to-copy code, but rather as a description or template of a solution for a recurring problem, which can be adapted to different contexts.

The best-known and most widely used patterns in the software industry are those from the *Gang of Four* (GoF), introduced by [Gamma et al. 1994]. In this work, the authors describe 23 patterns, classified into three main categories according to their purpose:

- Creational Patterns: Focus on managing and abstracting the process of object creation and instantiation, making systems more flexible regarding how objects are created.
- Structural Patterns: Deal with the composition of classes and objects to form larger and more complex structures, contributing to the system's organization.
- Behavioral Patterns: Involve the interaction and distribution of responsibilities among classes and objects, optimizing the communication between them.

This study focuses exclusively on creational patterns, which are fundamental to controlling how objects are initialized, a central task in many software systems. The creational patterns are described below:

- Singleton: Ensures that a class has at most a single instance and provides a global access point to that instance. It is commonly used to manage shared resources, such as database connections or logging services.
- Prototype: Allows the creation of new objects by copying an existing object, which serves as a prototype. This pattern avoids the cost of creating an object from scratch and is useful when the instantiation process is complex.
- Builder: Separates the construction of a complex object from its final representation, allowing the same construction process to produce different representations. It is beneficial for avoiding constructors with many parameters, enabling the incremental creation of the object.

[1]More information available at: `https://chatgpt.com/`.
[2]More information available at: `https://gemini.google.com/?hl=en`.
[3]More information available at: `https://copilot.microsoft.com/`.

- Factory Method: Defines an interface for creating an object, delegating the decision of which concrete class to instantiate to its subclasses. This pattern promotes low coupling between the involved classes.
- Abstract Factory: Provides an interface for creating families of related or interdependent objects without specifying their concrete classes.

3. Materials and methods

3.1. Scope

This study is guided by research questions that investigate the ability of LLMs to generate code that adheres to design patterns and the quality of this code. The research questions (RQs) are described below, accompanied by their respective justifications:

RQ1: To what extent can ChatGPT, Gemini, and Copilot correctly implement GoF's creational design patterns?

Justification: Considering that the literature indicates significant error rates in LLM responses, evaluating their ability to implement a fundamental concept such as design patterns correctly is the first step in assessing their viability.

RQ2: Is there a difference in complexity between the programs generated by LLMs and the reference programs[4]?

Justification: Code complexity directly impacts testing, comprehension, and maintenance efforts. One of the goals of design patterns is precisely to manage this complexity. Thus, even if an LLM correctly applies the pattern (RQ1), it is essential to investigate whether it does so in a simple manner or introduces unnecessary complexity.

RQ3: Is there a difference in maintainability between the programs generated by LLMs and the reference programs?

Justification: Maintainability is a fundamental quality attribute that directly impacts the project's cost throughout its lifecycle. If LLMs generate functional but hard-to-maintain code, their value as a productivity tool may be limited.

To investigate these questions, the following analysis factors were considered:

- Implementation correctness: evaluated qualitatively, this dimension verifies if the generated code meets the technical requirements of the design pattern and the specifications provided in the prompt. For this, the programs were individually inspected, looking for the following failures:
  - Code generation failure: the LLM generates no code as a response.
  - Pattern generation failure: the generated code does not follow or correctly implement the requested pattern.
  - Prompt failure: the code fully or partially disregards the provided instructions. This failure is subdivided into:
    - Incorrect names: attributes, variables, methods, or classes different from what was requested.
    - Unsolicited code: the model adds unrequested elements.

[4]Reference programs are source codes that serve as a basis for creating prompts and as a control group in the quality analysis.

        * Incorrect context: the code ignores the prompt.
- Complexity: Measured quantitatively by Halstead's complexity metrics [Halstead 1977], which consider the count of operators and operands. The applied metrics include:
    - Vocabulary of the program: number of unique terms needed to understand the code.
    - Length of the program: total sum of operators and operands.
    - Volume of the program: a measure of the informational content of the code, in bits.
    - Difficulty of the program: an estimate of the cognitive load to understand the code.
    - Effort: an estimate of the effort required to rewrite the code.
- Maintainability: Assessed using the maintainability index, a standardized metric from 0 to 100.
    - Values close to 100 indicate easier to understand, modify, and maintain code.
    - Values close to 0 indicate more complex and hard-to-maintain code.

3.2. Study design

The study was based on a comparative analysis between the code generated by three LLMs and a set of reference codes.

The selected versions were: ChatGPT 3.5, Gemini 1.5 Pro, and Copilot (2024 version), representing three of the most relevant and up-to-date models available. The code generated by each model was evaluated for implementation correctness, complexity, and maintainability.

To ensure a contextualized (rather than purely absolute) analysis, a control group composed of 50 human-written reference programs was established. This group provides a baseline for objective comparison with the code generated by the LLMs.

Each of the five investigated creational patterns (Singleton, Prototype, Builder, Factory Method, and Abstract Factory) was represented by 10 programs, totaling 50 in the control group.

3.3. Instrumentation and data collection

The evaluation of the LLMs first required constructing a set of human-written reference programs. These programs served both as a basis for creating the prompts and as the control group.

The collection of the programs involved a search in object-oriented programming textbooks and open-source repositories. When the code was not available in a textual format, the Editpad[5] tool was used for OCR extraction. Codes in other languages, such as Java or C++, were converted to Python using the CodeConverter[6] tool.

Each code was executed and manually inspected to ensure its correct functionality and faithfulness to the pattern's implementation. At the end of the process, 50 valid programs were selected (10 for each pattern).

[5] https://www.editpad.org/

[6] https://www.codeconvert.ai/

Based on these codes, 50 prompts were created, each describing the problem, explicitly requesting a specific pattern, and requiring the Python language as output.

3.4. Data analysis

The analysis was divided into two fronts:

- Qualitative (RQ1): Performed manually by inspecting the 150 codes generated by the models. The results were classified into three categories:
  - Critical failure: Encompasses programs that exhibited Code generation failure" or Pattern generation failure." In the study context, these were considered the most severe failures, as they compromise the central objective of the task, rendering the generated code unusable.
  - Partial failure: Includes programs that correctly applied the design pattern but incurred some form of "Prompt failure" (incorrect names, extra code, etc.). These programs were considered non-ideal but reusable, as the pattern was implemented and the deviations could be corrected with minor changes.
  - Ideal code: Represents programs that did not exhibit any of the listed failures. The generated code fully met the requested design pattern and the context and specifications described in the prompt.
- Quantitative (RQ2 and RQ3): Conducted with the Radon (v6.0.1)[7] tool, which was used to calculate the maintainability index and Halstead's metrics. This tool ensured uniformity in the metrics extraction, allowing fair comparisons among all 200 analyzed programs.

4. Results

This section presents the collected results, organized according to each of the three research questions that guided the study.

4.1. Capacity to apply design patterns

Table 1 provides an overview of the LLMs' performance for each design pattern. According to the results, 22.67% of code generation attempts on average resulted in a "Critical failure". In practice, more than one in five generated codes are unusable because they do not implement the requested pattern or are due to generation failures. Adding the partial failures (12.67%), the rate of imperfect code generation rises to 35.34%, indicating that human intervention is necessary and cannot be treated as an exception.

Table 1. Percentage distribution of ideal, partially failed, and critically failed codes, by design pattern

Design Pattern	Ideal	Partial failure	Critical failure
Singleton	70.00%	13.33%	16.67%
Prototype	56.67%	23.33%	20.00%
Builder	63.33%	10.00%	26.67%
Factory Method	60.00%	13.33%	26.67%
Abstract Factory	73.33%	3.33%	23.33%
Mean	64.67%	12.67%	22.67%

[7] https://radon.readthedocs.io/en/v6.0.1/

The analysis by design pattern reveals a variability in the LLMs' performance. The Abstract Factory pattern showed the highest success rate, with 73.33% of ideal codes. However, even in this more favorable scenario, the chance of obtaining a code with some failure (summing partial and critical) is 26.66%. In contrast, the Builder and Factory Method patterns proved to be the most challenging, with the highest critical failure rate (26.67%), where more than a quarter of the code generations cannot be used.

The Prototype pattern deserves special attention because it did not have the highest critical failure rate; it recorded the lowest success rate (56.67%) and the highest partial failure rate (23.33%). This result suggests that LLMs produce code that appears correct but contains subtle deviations from what was requested.

Table 2 details the individual performance of each LLM, revealing distinct behavioral profiles during the generation of code adhering to the different design patterns.

Table 2. Percentage distribution of ideal, partially failed, and critically failed codes, by design pattern and LLM

Design Pattern	LLM	Ideal	Partial Failure	Critical Failure
Singleton	ChatGPT	70.00%	10.00%	20.00%
	Gemini	60.00%	20.00%	20.00%
	Copilot	80.00%	10.00%	10.00%
Prototype	ChatGPT	60.00%	30.00%	10.00%
	Gemini	80.00%	20.00%	0.00%
	Copilot	30.00%	20.00%	50.00%
Builder	ChatGPT	60.00%	10.00%	30.00%
	Gemini	80.00%	0.00%	20.00%
	Copilot	50.00%	20.00%	30.00%
Factory Method	ChatGPT	80.00%	20.00%	0.00%
	Gemini	40.00%	10.00%	50.00%
	Copilot	60.00%	10.00%	30.00%
Abstract Factory	ChatGPT	80.00%	10.00%	10.00%
	Gemini	90.00%	0.00%	10.00%
	Copilot	50.00%	0.00%	50.00%

Copilot proved to be the least reliable, with critical failure rates of 50% in the Prototype and Abstract Factory patterns. Its best performance (80% success) occurred only in the Singleton pattern, indicating possible limitations in generating more complex logic.

Gemini showed a polarized performance, achieving good results in the Builder and Abstract Factory patterns (80% and 90% success, respectively), but unsatisfactory performance in the Factory Method, with a 50% critical failure rate.

ChatGPT stood out for its consistency. Despite not reaching Gemini's performance, it did not have many critical failures. Its critical failure rate remained between 0% and 30%, reaching 0% in the Factory Method, which was its best result. This profile suggests a generalist model with a lower risk of serious failures.

4.2. Complexity analysis

The second research question sought to understand if there is a difference between the complexity of the programs generated by the LLMs and the reference programs. To answer this question, the averages of Halstead's complexity measures were calculated for each LLM and the human-produced codes (reference code), as presented in Table 3.

According to Table 3, the behavior of the LLMs is dependent on the design pattern, with no model being consistently "more" or "less" complex. An interesting result

Table 3. Quantitative analysis of the complexity of programs generated by LLMs and humans

Design Pattern	LLM	Vocabulary	Length	Volume	Difficulty	Effort
Singleton	ChatGPT	6	7	18.75	0.82	15.65
	Gemini	8.25	11.25	35.88	1.07	45.11
	Copilot	4.89	5.56	12.79	0.69	8.99
	Reference	7.1	10.67	25.38	0.98	34.29
Prototype	ChatGPT	0.3	0.3	0.48	0.05	0.24
	Gemini	0.6	0.6	0.95	0.1	0.48
	Copilot	0.3	0.3	0.48	0.05	0.24
	Reference	2.7	2.7	4.88	0.45	3.22
Builder	ChatGPT	2.78	3.33	11.14	0.28	10.88
	Gemini	-	-	-	-	-
	Copilot	0.78	1	2.81	0.25	6.32
	Reference	4.8	5.7	20.51	0.5	21.87
Factory Method	ChatGPT	8.2	14	45.42	0.75	34.87
	Gemini	4	5.89	13.58	0.54	8.79
	Copilot	3.89	5.56	13.62	0.47	7.99
	Reference	3.7	4.7	9.87	0.57	6.86
Abstract Factory	ChatGPT	3.6	4.2	9.06	0.52	5.77
	Gemini	4.56	5.67	15.37	0.46	8.77
	Copilot	3.1	4.2	9.92	0.39	5.99
	Reference	5.2	6.5	20.95	0.52	23.81

was observed with the Prototype pattern, where all LLMs generated code with lower complexity than the reference programs. The Difficulty (0.05 to 0.1) and Effort (0.24 to 0.48) metrics for the LLMs were an order of magnitude smaller than those of the reference (Difficulty of 0.45 and Effort of 3.22). This result raises questions: *does the simplicity of LLM-generated code indicate superior model efficiency or simply reveal 'bloat' in human-written reference code?*.

A similar, though less pronounced, trend was observed in the Abstract Factory and Builder patterns. For the Abstract Factory, all LLMs produced less complex code than the reference in almost all metrics. In the Builder, both ChatGPT and Copilot also generated less complex code. It is important to note the absence of metrics for Gemini in the Builder pattern; this occurred because the code it generated for this pattern did not have enough operators or operands for calculation, resulting in null values. This finding suggests incomplete code generation by Gemini for this design pattern.

The complexity analysis revealed unpredictable behavior from the LLMs, especially in the Singleton and Factory Method patterns. This unpredictability can be observed in two ways. First, the results vary among the models for the same task. In the case of Singleton, Gemini generated the most complex code, while Copilot produced the simplest, showing opposite behaviors for the same problem. Second, in the anomalous behavior of a single model. ChatGPT stood out negatively for the Factory Method by generating code with disproportionately higher Volume (45.42) and Effort (34.87) than all other LLMs, including the human reference.

Given these results, it is believed that there are differences in complexity, but not consistently. The complexity of the code generated by an LLM is not a characteristic of the model itself, but rather of the "model-pattern" pair. The analysis reveals that LLMs often generate less complex code than human-produced code, but this "simplicity" should be viewed with suspicion, as it may indicate a superficial implementation. The complexity obtained with the codes generated by ChatGPT for the Factory Method is the most critical finding, as it suggests that a developer cannot have guarantees about the simplicity of the code generated by this LLM.

4.3. Maintainability analysis

The third and final research question sought to investigate whether there is a significant difference between the maintainability of the programs generated by the LLMs and the reference programs. Table 4 presents the findings.

Table 4. Maintainability index among LLMs and human-written codes by design pattern

Design Pattern	LLM	Maintainability Index
Singleton	ChatGPT	89.17
	Gemini	77.63
	Copilot	91.02
	Reference	75.48
Prototype	ChatGPT	97.08
	Gemini	95.07
	Copilot	98.5
	Reference	78.36
Builder	ChatGPT	88.95
	Gemini	-
	Copilot	97.12
	Reference	71.59
Factory Method	ChatGPT	76.79
	Gemini	77.49
	Copilot	85.56
	Reference	77.17
Abstract Factory	ChatGPT	78.59
	Gemini	76.01
	Copilot	82.25
	Reference	80.77

In most scenarios, the LLMs surpassed the reference programs in terms of maintainability. This result was particularly evident in the Singleton, Prototype, and Builder patterns. In the case of Prototype, all LLMs obtained a maintainability index above 95, while the reference reached only 78.36. In the Builder, Copilot achieved a maintainability index of 97.12, higher than the reference's value of 71.59.

Copilot was the best-performing model in terms of maintainability in most patterns. However, the Abstract Factory pattern presented an exception, as the human reference (maintainability index = 80.77) outperformed both ChatGPT (78.59) and Gemini (76.01), being only marginally surpassed by Copilot (82.25).

In this sense, there is a difference in maintainability, which is a general advantage for the LLMs. The models, especially Copilot, demonstrated a strong ability to generate structurally easier code to maintain than the reference code. However, this advantage is not absolute. The result of the Abstract Factory raises the hypothesis that for patterns of high structural complexity, a well-crafted human implementation may still be superior, challenging the apparent superiority of the LLMs.

5. Limitations

It is important to acknowledge the limitations of this study to contextualize the results and guide future research properly. The main identified limitations are presented below.

Firstly, the scope of the investigation was restricted to creational design patterns. Thus, the results obtained regarding the capability of LLMs and the quality of the generated code cannot be generalized to other pattern categories, such as structural and behavioral. These categories involve different types of complexity, such as the composition

and organization of objects or the coordination of their interactions, situations in which the performance of LLMs may differ.

Secondly, the study was conducted exclusively using Python. As the proficiency of LLMs varies depending on the language used, the presented results may not be replicable in contexts involving other languages, such as Java, C++, or JavaScript.

The third limitation refers to the lack of variation in the prompts used. For each scenario, a single prompt was adopted, without exploring how different formulations or levels of detail could impact the results. Considering that prompt engineering is a known factor influencing LLMs' behavior significantly, this variable represents an important gap to be investigated in future studies.

Finally, a researcher performed a qualitative analysis of the code's correctness (related to RQ1). Despite following predefined criteria, this approach may have introduced a subjective bias in classifying the codes into the "Ideal code", "Partial failure", and "Critical failure" categories. Future studies could employ multiple evaluators to increase the reliability of the results.

These limitations do not invalidate the findings of this work. However, they define the boundaries of its applicability and signal promising directions for subsequent research in automatic code generation with LLMs.

6. Conclusions

Given the growing adoption of LLMs in software systems development and the lack of studies on their reliability in generating code that adheres to design patterns, the purpose of this study was to evaluate the ability of the LLMs ChatGPT, Gemini, and Copilot to generate code compliant with creational design patterns. To this end, a study was conducted that analyzed 150 programs generated by the LLMs, evaluating them on three dimensions: the correctness of the pattern's implementation, complexity, and maintainability, in comparison with a set of 50 reference programs.

The results indicate that, although LLMs can apply design patterns in most cases, they still lack consistency, presenting a critical failure rate of 22.67%. This failure rate suggests that adopting these models to facilitate the application of design patterns may be premature. Regarding code quality, no differences in complexity were found compared to human-produced code. Additionally, the code generated by LLMs tended to have a higher maintainability index, suggesting a potential for generating structurally simple code. It is important, however, to contextualize these results within the study's limitations, which were restricted to creational patterns in the Python language and did not explore variations in *prompt* engineering.

Based on the results and limitations, several opportunities for future investigations emerge. This study is suggested to be replicated with structural and behavioral patterns to verify if the LLMs' performance is maintained. Conducting experiments in other languages, such as Java or C++, would also be of great value. Additionally, future research could focus on the impact of prompt engineering on the quality of the generated code and investigate the causes of the LLMs' difficulties with specific patterns. Finally, it would be interesting to deepen the analysis of the high maintainability found, verifying if it represents better quality code.

Artifact availability

To support future work that addresses the identified limitations, a lab package has been prepared and is available at: `https://bit.ly/4fAEVrx`.

Acknowledgments

The authors acknowledge the Fundação Araucária de Apoio ao Desenvolvimento Científico e Tecnológico do Estado do Paraná (Fundação Araucária), PROPP/UFGD - SIGProj No. 322855.1174.8276.11032019, and the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), under processes no. 407879/2023-4, 302937/2023-4, and 442262/2024-8.

References

Feuerriegel, S., Hartmann, J., Janiesch, C., and Zschech, P. (2023). Generative ai. *Business & Information Systems Engineering*. Accessed: 17 Mar. 2024.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, Massachusetts, USA, 1 edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2007). *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Bookman, Porto Alegre, Rio Grande do Sul, Brasil, 1 edition.

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

Jamil, M. T., Abid, S., and Shamail, S. (2025). Can llms generate higher quality code than humans? an empirical study. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 478–489. IEEE.

Kabir, S., Kou, B., Udo-Imeh, D. N., and Zhang, T. (2024). Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions. Accessed: 21 Mar. 2024.

Li, J. et al. (2023). Acecoder: Utilizing existing code to enhance code generation. *arXiv preprint arXiv:2303.17780*.

Liu, H. et al. (2023). Autotestgpt: A system for the automated generation of software test cases based on chatgpt. Available at SSRN 4584792.

Nguyen-Duc, A. et al. (2023). Generative artificial intelligence for software engineering–a research agenda. *arXiv preprint arXiv:2310.18648*.

Riehle, D. (2011). Transactions on pattern languages of programming ii. In *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, Germany. Accessed: 13 Oct. 2024.

Simões, I. R. d. S. and Venson, E. (2024). Evaluating source code quality with large language models: a comparative study. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*, pages 103–113.