Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 33, Num. 3 (2026) 90-108

RESEARCH ARTICLE

Analyzing Correctness, Memory Consumption, CPU Usage, and Execution Time of the LLMs Llama-3.3-70B Versatile and Gemma2-9B Instruct

Analisando Corretude, Consumo de Memória, Uso de CPU e Tempo de Execução dos LLMs llama 3.3-70B Versatile e Gemma2-9B Instruct

Rodrigo Andrade[1]* and Jackson Lima[1]

Abstract: Although the rapid advancement of Large Language Models (LLMs) has significantly transformed software development, enabling the automation of tasks such as code generation, testing, and documentation, concerns remain regarding the quality, efficiency, and security of automatically generated solutions, especially in relation to computational resource usage such as memory, CPU, and execution time. This scenario is intensified by the growing demand for digital sustainability and the need to minimize operational costs. This work investigates the efficiency and correctness of code generated by the LLMs Llama-3.3-70B Versatile and Gemma2-9B Instruct when solving Rosetta Code tasks in Python, comparing them with human implementations. The analyses address multiple aspects, including code correctness, clarity, memory consumption, CPU usage, and execution time, thus revealing the limitations and strengths of automatic solutions compared to human-developed code. The results suggest that, although LLMs are capable of quickly generating functional solutions, they still lack the algorithmic optimizations necessary to match the efficiency of manually written code, particularly regarding resource utilization. However, for routine or well-known tasks, automatic solutions can be highly competitive. This study highlights the importance of multidimensional evaluations and demonstrates that, if combined with human review and good engineering practices, the careful adoption of LLMs can enhance software development processes.
Keywords: LLM — code correctness — memory consumption — CPU usage — execution time

Resumo: O avanço dos Modelos de Linguagem de Grande Porte (LLMs) transformou significativamente o desenvolvimento de software, permitindo automação de tarefas como geração de código, testes e documentação. Apesar desses avanços, ainda existem debates sobre a qualidade, eficiência e segurança das soluções geradas automaticamente, principalmente no que diz respeito ao uso de recursos computacionais como memória, CPU e tempo de execução. Esse cenário é amplificado pela crescente demanda por sustentabilidade digital e pela necessidade de minimizar custos operacionais. Este trabalho investiga a eficiência e a corretude de códigos gerados pelos LLMs Llama-3.3-70B Versatile e Gemma2-9B Instruct ao resolver tarefas do Rosetta Code em Python, comparando-os com implementações humanas. As análises abordam diversos aspectos, incluindo execução correta, clareza, consumo de memória, uso de CPU e tempo de execução, evidenciando limitações e pontos fortes das soluções automáticas frente às humanas. Os resultados mostram que, embora LLMs apresentem alta capacidade para gerar códigos funcionais de maneira rápida, ainda carecem de otimizações necessárias para alcançar a mesma eficiência de soluções elaboradas manualmente, especialmente quanto ao uso de recursos. No entanto, nota-se que, em tarefas rotineiras ou bem conhecidas, as soluções automáticas podem ser altamente competitivas. Por fim, este estudo enfatiza a importância de avaliações sob diferentes perspectivas, apontando que a adoção criteriosa de LLMs pode potencializar o desenvolvimento de software se combinada com revisões humanas e boas práticas de engenharia.
Palavras-Chave: LLM — corretude de código — consumo de memória — uso de CPU — tempo de execução

[1] *Universidade Federal do Agreste de Pernambuco (UFAP), Brazil*
*__Corresponding author__: rodrigo.andrade@ufape.edu.br
DOI: http://dx.doi.org/10.22456/2175-2745.153834 • **Received:** 25/02/2026 • **Accepted:** 18/05/2026

1. Introduction

The rapid advancement of Large Language Models (LLMs) has revolutionized the field of software engineering, becoming increasingly present in stages such as code writing, test automation, and documentation generation [1]. Models such as ChatGPT [2], Gemini [3], Llama [4], and Claude [5] demonstrate the ability to understand natural language instructions and provide programming solutions for a wide range of challenges, increasing productivity, democratizing access to development, and opening new perspectives for the automation of complex tasks [6, 7]. Nevertheless, questions regarding the quality, security, and efficiency of the generated code remain under debate, especially in production environments and critical applications [8].

In parallel, the need to measure and minimize the use of computational resources such as memory, CPU, and energy has become central for researchers and practitioners in the field [9, 10]. The growing demand for digital sustainability, associated with the expansion of large-scale software use and the need to reduce costs, requires continuous attention to resource consumption during program execution [8]. Recent studies demonstrate that the choice of language, implementation style, and adopted practices have a direct impact on the energy and hardware efficiency of computational systems [9, 10]. In the context of code generation with LLMs, these challenges are accentuated, as automated systems may reproduce less efficient practices or generate solutions that are inefficient from a computational standpoint [11].

In this context, the present work investigates the efficiency and correctness of code generated by large language models, comparing their performance with human solutions. Two models are utilized, Llama-3.3-70B Versatile and Gemma2-9B Instruct, which are evaluated based on the automatic generation of Python scripts for a set of tasks from Rosetta Code [12]. The analyses include dimensions such as correct code execution, solution clarity, memory consumption, CPU usage, and execution time, enabling the identification of limitations, strengths, and gaps in automated solutions compared to human-developed alternatives [7, 13].

By comparatively addressing LLM-generated code and their human-written counterparts, regarding both functional and performance aspects, this study contributes to the understanding of the potentials and limits of LLMs in programming automation. The results may assist developers and researchers in a more judicious adoption of these tools, signaling advantages, risks, and practical consequences from the perspective of efficiency and digital sustainability [8, 1]. Thus, it is expected to favor the responsible use of LLMs in Software Engineering, stimulating technical advances and the development of more sustainable practices in the contemporary scenario.

The results obtained throughout this work indicate that, although LLMs are capable of generating functional code and, in many cases, correctly implement the logic of the proposed problems, there are still significant limitations regarding the efficiency of computational resource use and the adoption of programming best practices [7]. In several situations, automated solutions exhibited higher memory and CPU consumption compared to human implementations from Rosetta Code, primarily due to the absence of algorithmic optimizations and the reproduction of less efficient constructs [1, 8]. On the other hand, LLMs demonstrated a remarkable capacity for rapid generation of functional scripts and for handling most test cases, excelling in more routine tasks. These findings reinforce the importance of evaluating not only correctness but also practical performance when adopting automated solutions in software development [13, 7].

The remainder of this work is organized as follows. Section 2 details the study settings while Section 3 explains our findings. Moreover, Section 4 discusses related work and Section 5 presents the final remarks.

2. Methodology

In this section, we explain the design of our study. Thus, Section 2.1 presents the research questions and the selected metrics. In Section 2.2, we explain our prompting strategy with LLMs, while Section 2.3 details Rosetta Code. Finally, Sections 2.4 and 2.5 detail, respectively, the sample of tasks selected for this work and the chosen LLMs.

2.1 Research Questions and Metrics

Table 1 presents the research questions along with the corresponding metrics used to address them. **RQ1** explores whether the llama-3.3-70b-versatile (Llama) and gemma2-9b-it (Gemma) models can correctly generate Python implementations in response to a textual prompt. Answering **RQ1** is fundamental to evaluate the feasibility of delegating the implementation of well-known problems to LLMs. To assess this capability, we adopted the Percentage of Correct Implementations (`PCI`) metric, which measures the proportion of implementations generated by the LLMs that are syntactically correct and functional for the provided prompt. Thus, a correct response is an implementation that, in addition to being free of syntax errors, also presents the expected logic to solve the proposed problem.

To help measuring correctness, we define the dimensions of clarity, procedural logic, and omissions, which we operationalize through a standardized evaluation rubric. **Clarity** is defined by the structural readability of the implementation, specifically assessing the use of descriptive identifiers (e.g., variable and function names), adherence to consistent indentation, and the absence of redundant code segments that could obscure the algorithm's intent. Additionally, **Procedural Logic** is measured by the degree of alignment between the code's control flow and the specific mathematical or logical steps provided in the prompt's problem description. Also, **Omissions** is operationalized as the absence of required functional components, such as failing to handle specific edge cases, missing iterations over the mandated range, or omitting requested return formats. Table 2 summarizes these dimensions' definitions.

Table 1. Research Questions and Metrics

Research Question	Metric
RQ1: What is the percentage of correct implementations that Llama 3 and Gemma 2 can generate from a textual prompt?	Percentage of Correct Implementations (`PCI`)
RQ2: Are human-written implementations more memory-efficient than those generated by Llama 3 and Gemma 2?	Memory consumption in `bytes`
RQ3: Are human-written implementations more CPU-efficient than those generated by Llama 3 and Gemma 2?	CPU usage in `percentage`
RQ4: Are human-written implementations more time-efficient than those generated by Llama 3 and Gemma 2?	Execution time in `seconds`

Table 2. Summary of Operational Definitions for Qualitative Analysis

Dimension	Operational Definition
Clarity	Use of descriptive naming (identifiers), consistent code structure, and absence of redundant logic that obscures intent.
Procedural Logic	Direct alignment with the step-by-step mathematical or logical instructions provided in the prompt's problem description.
Omissions	Absence of required functional blocks, failure to handle edge cases, or missing iterations over the stipulated input range.

In particular, we measure `PCI` through a validation process combining automated testing and manual qualitative review. Initially, each LLM-generated script undergoes an automated executability check to ensure it was free of syntax errors and could run within the established measurement environment. Following this, we conduct a manual analysis to evaluate the correct output and **Procedural Logic**, comparing the model's results against the expected mathematical or logical outcomes defined in the problem description.

To mitigate subjectivity in the manual analysis, the first author conducts the evaluation by following the standardized rubric detailed in Table 2. To ensure the reliability of these assessments, we execute a cross-verification protocol: the second author independently reviews a subset of the evaluations (approximately 20% of the tasks) to check for consistency in the application of "Partial" and "Incorrect" classifications. In this context, we resolve, any discrepancies in the qualitative ratings for dimensions such as **Clarity** or **Omissions** through collaborative discussion between the authors until they reach a consensus.

Regarding **RQ2**, our goal is to investigate whether human-written implementations consume more or less memory than those generated by LLMs. This is important because, in addition to being correct, implementations generated by LLMs must be practical for real-world use. Therefore, to answer this research question, we measure memory consumption in `bytes`.

Furthermore, **RQ3** investigates the efficiency of LLM-generated implementations in terms of CPU usage compared to their human-written counterparts. Consistent with the logic of **RQ2**, the practical viability of these implementations depends on reasonable CPU utilization. Consequently, we quantify this aspect by measuring the percentage of CPU consumed during the algorithm's execution.

Last but not least, **RQ4** analyzes efficiency regarding execution time. Thus, our objective is to compare the execution time of algorithmic implementations generated by LLMs with those written by humans. To answer this research question, we measure the execution time in `seconds`.

2.2 Providing Instructions to LLMs

To solicit responses from the models, we designed a prompt template divided into three main parts. The first part of the prompt introduces the general context of the task, positioning the model in the role of a software developer tasked with solving a computational problem. In the second part, we provide a detailed description of the problem to be solved and, where applicable, input and output examples. Finally, the third part clearly specifies the expected response from the model, requesting functional and objective Python code without any additional text or explanation. Additionally, all prompts were developed in English, as this language yields better performance and is natively optimized by both Llama and Gemma. Listing 1 illustrates the complete structure of a prompt used in the experiments.

First, the prompt establishes the task context by explicitly instructing the model to position itself as a software developer responsible for solving a computational problem (lines 1–2 of Listing 1). This initial section guides the model to understand

Listing 1. Prompt example

```
1 Context:
2 You are a software developer. Your task is to solve a computational problem based on the
      description provided below...
3
4 Problem Description:
5 Let P(n) be the sum of the proper divisors of n where
6 the proper divisors are all positive divisors of n other
7 than n itself.
8
9   if P(n) < n then n is classed as deficient (OEIS A005100).
10  if P(n) == n then n is classed as perfect (OEIS A000396).
11  if P(n) > n then n is classed as abundant (OEIS A005101).
12
13 Calculate how many of the integers 1 to 20,000 (inclusive) are in each of the three classes.

14
15 Example:
16  6 has proper divisors of 1, 2, and 3.
17  1 + 2 + 3 = 6, so 6 is classed as a perfect number.
18
19 Task:
20 Write a function in python that solves the described problem.
21 - The function should receive input and return output as appropriate for the task.
22 - Provide only the function code, with no additional explanations or comments.
```

that its role is to address a technical challenge described in a structured manner.

Subsequently, in the ”Problem Description” section, the prompt presents the formal definition of the mathematical problem[1] (lines 4–14). Specifically, it introduces the function $P(n)$, defined as the sum of the proper divisors of an integer n (lines 5–6), clarifying that a proper divisor is any positive divisor of n excluding n itself (lines 6–7). Then, lines 9–11 clearly establish the three possible types of classification for the integer based on the relationship between $P(n)$ and n: deficient, perfect, or abundant, each referenced by OEIS identifiers.

After defining the categories, the prompt specifies the central objective in line 13: to determine, within the range of 1 to 20,000 (inclusive), how many integers belong to each of the three classes. This directs the solution to iterate through the entire stipulated range, apply the analysis to each number, and count the frequency of each classification.

The ”Example” section (lines 16–18) illustrates the concept using the number 6. First, it details its proper divisors (line 17), performs the summation (line 18), and subsequently classifies the number as perfect. This example provides a concrete input-output case and serves as a guide for the correct logic of the solution.

Finally, in the ”Task” section (lines 20–23), the prompt clearly demands a practical Python solution: it requests the model to write a function capable of solving the problem as described. In lines 22–23, it further specifies that the function should receive input and return output as appropriate for the task and, explicitly, requires that the provided response consists only of the function code, without explanatory text or additional comments.

Overall, the prompt is structured to guide the model in understanding the context, internalizing the mathematical requirements of the problem, assimilating a representative example, and finally, producing an objective and functional Python implementation to perform the counting of the three numerical classes in the range of 1 to 20,000.

2.3 Rosetta Code

Rosetta Code presents a wide variety of programming challenges and their solutions in several languages [12]. The ”Solutions by Programming Task” category[2], in particular, functions as a comprehensive index of these challenges, allowing users to compare and contrast different programming paradigms and syntaxes. This can be particularly useful for those learning a new language, as it offers a practical way to visualize how familiar concepts are implemented in unfamiliar environments.

The platform organizes tasks into both alphabetical and thematic categories, making it easy for users to locate what they are looking for. After selecting a task, the user is directed to a page that includes the problem description, followed by a series of solutions in different programming languages. This

[1] ⟨https://rosettacode.org/wiki/Proper_divisors⟩

[2] ⟨https://rosettacode.org/wiki/Category:SolutionsbyProgramming_Task⟩

side-by-side comparison is the foundation of Rosetta Code's utility, as it enables a deep understanding of the nuances and strengths of each language. Furthermore, the collaborative nature of the site ensures that new tasks and solutions are constantly added, keeping the content relevant and up to date.

Rosetta Code provides its own description for each programming task, which we incorporated into our prompt design strategy (as illustrated in Listing 1). For instance, Rosetta Code offers implementations of Dijkstra's Algorithm[3] in 67 different languages, ranging from the widely used Java to less common languages such as Phix. Consequently, in this work, we use the human-written Python implementations available on Rosetta Code as our reference.

2.4 Selected Tasks

To compose our sample of tasks from Rosetta Code, we first sought well-defined computational problems that require the implementation of a specific algorithm or a logical procedure to process data and achieve a verifiable result. Each task must be a self-contained challenge with clear rules and a defined objective, whose solution is attainable through computation.

Specifically, we divided the sample selection criteria into three categories: (i) Mathematics and Number Theory, (ii) Algorithms and Logic Problems, and (iii) Data and String Processing. The first category requires the application of mathematical definitions and formulas in the code to classify numbers or calculate values. The second category focuses on implementing known algorithms or developing a logical procedure to traverse a state space, find a path, or solve a logic challenge. Finally, the third category involves receiving input data (such as text or files), processing it according to a set of rules, and generating a specific output format or result. Following these criteria and their respective categories, we selected 23 tasks from Rosetta Code. Table 3 illustrates these tasks and their respective categories.

2.5 Large Language Models (LLMs)

This study considers two LLMs for comparative analysis: llama-3.3-70b-versatile [14] and gemma2-9b-it [15], which are detailed in Sections 2.5.1 and 2.5.2, respectively.

We guide the selection of the these models for this comparative analysis by technical and methodological criteria. Primarily, this choice allows for a direct comparison between two distinct architectural philosophies and scales: the high-parameter Llama-3.3-70B developed by Meta and the more compact, efficiency-focused Gemma2-9B developed by Google. Methodologically, we justify the use of these models by their shared integration into the GroqCloud platform, which provides an identical computing environment for both architectures. This consistency is crucial for the validity of our execution time and resource consumption measurements.

[3] ⟨https://rosettacode.org/wiki/Dijkstra%27s_algorithm⟩

2.5.1 Llama 3.3-70B Versatile (`llama-3.3-70b-versatile`)

The llama-3.3-70b-versatile (Llama) is a Large Language Model (LLM) developed by [16], designed to generate coherent and contextually appropriate text based on user-provided prompts. Its architecture is based on transformers, utilizing self-attention mechanisms that allow the model to weight the relative importance of different words and tokens in a sequence, according to their relevance to the current context [16].

Like other state-of-the-art LLMs, Llama processes information in the form of tokens. These tokens correspond to text fragments—which can be words, subwords, or characters—that are converted into numerical vectors and fed into the neural network. The model processes sequences of these tokens to understand and generate natural language. Each generated response is constructed by iteratively predicting the next token based on previous ones, ensuring the production of cohesive texts aligned with the context of the provided prompt [17, 16].

The training of Llama involved large volumes of textual data from multiple sources, aiming to maximize the model's generalization and versatility across diverse language tasks. The fine-tuning process included the use of techniques such as Reinforcement Learning from Human Feedback (RLHF), where human evaluators rank model-generated responses, providing feedback to improve response quality [18, 17]. Furthermore, the model is available through various interfaces, enabling researchers and developers to input text prompts and evaluate the generated responses.

2.5.2 Gemma2-9B Instruct (`gemma2-9b-it`)

The gemma2-9b-it (Gemma) is a Large Language Model (LLM) developed by Google [19], aiming to provide coherent, safe, and context-aligned responses to the user. Despite being focused on textual processing, the model was designed for excellence in tasks such as text and code understanding, generation, and manipulation, with a particular emphasis on safety, open-source availability, and efficiency [19].

Gemma's operation is based on deep neural networks trained on vast volumes of data from diverse sources, including technical texts, documentation, code, and natural language. Like other LLMs, it employs the concept of tokens, which are small units of text (words, subwords, or characters) converted into numerical vectors and processed in sequence by the model [19]. Through iterative predictive token generation, the model can produce cohesive and informative texts, aligning the output with the context of the provided prompt.

In code generation tasks, the model demonstrates the ability to interpret, explain, and produce code in several widely used languages, including Python, Java, and C++. The model also features an extended context window, being capable of processing extensive inputs, which facilitates the analysis of long documents, large code blocks, and tasks that require long-term comprehension [19].

Table 3. Selected Rosetta Code Tasks and Their Categories

Mathematics & Number Theory	Algorithmic & Logic Puzzles	Data & String Processing
Abundant deficient and perfect number classifications	ABC problem	English cardinal anagrams
Abundant odd numbers	Countdown	Exactly three adjacent 3 in lists
Additive primes	Cycle detection	Extended straddling checkerboard
Almkvist-Giullera formula for pi	Dijkstra algorithm	Last Friday of each month
Attractive Numbers	Free polyominoes enumeration	
CORDIC	100 doors	
Digit fifth powers	Iterators	
Discrete Fourier transform	Long stairs	
Find the intersection of two lines		
Largest product in a grid		
Magic numbers		

2.5.3 Accessing Models via GroqCloud

All experiments conducted in this study were carried out using the GroqCloud platform, an accelerated computing service that provides access to several Large Language Models (LLMs), including Llama and Gemma, via an API [20]. The choice of GroqCloud is justified by its simplified interface for querying multiple high-performance models and the possibility of equal access to different architectures.

Access to the GroqCloud API is performed via REST endpoints, using individual authentication tokens provided to the user. Requests can be made in various programming languages, as the API follows the OpenAI-compatible standard with JSON payloads.

Listing 2 demonstrates an example of a Python request performed to obtain a response from the llama-3.3-70b-versatile model. The procedure for the gemma2-9b-it model is analogous, requiring only the modification of the model identifier.

Listing 2. Example of access to the API GroqCloud

```
1  from groq import Groq
2
3  "provider": "groq_cloud"
4  "api_key": "chave_da_api"
5
6  def get_llm_response():
7   client = Groq(api_key=api_key)
8
9   chat_completion = client.chat.completions.
        create(
10   messages=[
11    {
12     "role": "user",
13     "content": "Explain the Bubble Sort
           Algorithm"
14    }
15   ],
16   model="llama-3.3-70b-versatile",
17   temperature=0,
18  )
19  response = chat_completion.choices[0].
        message.content
20  print(response)
```

Initially, the `Groq` class is imported from the appropriate library (line 1), which is necessary to prepare the code's execution environment. Subsequently, lines 3 and 4 present basic parameters for access configuration, including the service provider (`groq_cloud`) and an API key that must be provided by the user.

The core of the operation lies in the `get_llm_response` function (line 6), which encapsulates the entire automated process. Within this function, the Groq client is instantiated with the previously defined API key (line 7), allowing for secure authentication with the service.

The actual creation of the request occurs from line 9 onwards, using the `client.chat.completions.create` method to de-

fine the message and query parameters. The `messages` list (line 10) includes a dictionary (lines 11–14) representing the user message, composed of the role (`"role": "user"`, line 12) and the content of the request (`"content": "Explain the Bubble Sort Algorithm"`, line 13).

Next (lines 15–17), important arguments for the API are defined. First, the model to be used (`"llama-3.3-70b-versatile"`, line 16). Second, the `temperature` parameter, which controls the randomness of the response (line 17). After sending the request, the response returned by the model is stored in the `response` variable (line 19), being extracted directly from the API return object. Finally, the generated response is presented to the user through the `print(response)` command (line 20).

This example objectively demonstrates how to structure programmatic access to GroqCloud, perform queries to the available models, and extract responses generated dynamically from prompts.

2.6 Measurement Procedure

We developed a script to measure the performance of a Rosetta Code task implementation through the analysis of three metrics: memory usage, execution time, and CPU consumption. To achieve this, we utilized a decorator that wraps the task implementation, adding measurement functionalities to the process. Specifically, whenever the decorated task is called, the script executes the function one hundred times in a loop to obtain an average of the performance characteristics and to smooth out any random fluctuations in system performance.

In each of the one hundred executions, the script records three metrics. First, it monitors memory consumption by taking a snapshot of all memory allocated by the program immediately before and after the task execution. The difference between these two snapshots reveals the amount of memory utilized by the execution. Simultaneously, it records the exact time at the start and end of the execution, enabling the calculation of the total time spent. For CPU consumption, it measures user and system times at the beginning and end of the function, allowing the determination of the total CPU effort employed during the execution period.

After completing the one hundred executions, the script processes the collected data, calculating the averages for memory consumption, execution time, and CPU utilization from the gathered measurement lists. These average values are printed to the console, providing a summary of the task's performance. Additionally, the script saves a detailed report in a text file, which includes not only the calculated averages but also the complete lists of all individual memory and CPU measurements for each of the one hundred executions, enabling a more in-depth analysis. For the experiment, we also configured the script to execute the tasks one hundred times.

Finally, the environment used to perform the measurements consists of an AMD Ryzen 7-5800H processor, 32 GB of RAM, and the Ubuntu operating system.

3. Results

In this section, we present and discuss the main findings obtained from the conducted experiments. Thus, Section 3.1 addresses the correctness analysis of the solutions, evaluating criteria such as adherence to the problem statement, executability, output accuracy, procedural logic, and clarity. Subsequently, Section 3.2 details the results regarding memory consumption, while Section 3.3 presents the analysis of CPU usage. Section 3.4 explores the execution time of the implementations. Finally, Section 3.5 provides an integrated discussion of the observations extracted from the previous analyses, highlighting the identified implications and limitations.

3.1 Correctness

We evaluate the correctness of the solutions using the dimensions and standardized rubric established in Section 2.1 (Table 2). In this context, Tables 4 and 5 illustrate the results for correctness.

Each column in the table indicates a key aspect of correctness, allowing us not only to identify if the code "compiles and runs" but also to assess whether the response effectively solves the proposed problem according to the functional criteria required by the literature. Results marked as "Yes" indicate full compliance with the criterion, "No" indicates failure, and "Partial" reflects incomplete or partially correct responses.

A review of the table rows shows that, for some problems, both models produced fully correct solutions, while in others, partially correct responses or significant omissions are frequent. It is noteworthy that success rates are not homogeneous across models or task types, highlighting that the ability of LLMs to generate functional code is still, in part, dependent on prompt clarity, degree of difficulty, and the algorithmic nature of the task. Thus, the table consolidates the evaluation of **RQ1** and allows for a transparent discussion of the current capabilities of LLMs for automated programming tasks.

Additionally, a quantitative analysis of the "Correct Output" criterion (Table 6) measures the actual performance of the models in terms of successes, errors, and partial responses. Llama-3.3 generated fully correct outputs in 10 out of the 23 evaluated tasks (43%), presented partially correct responses in 9 (39%), and incorrect ones in 4 (17%). In contrast, Gemma2 obtained 6 fully correct responses (26%), 6 partially correct ones (26%), and 11 incorrect responses (48%).

As a response to **RQ1**, we can state that, although they have not yet reached ideal correctness rates, LLMs are approaching practical utility in automated programming, especially when considering partially correct solutions, which can be reviewed and adjusted with relative ease by human developers.

3.2 Memory consumption

This section seeks to answer the second question from Table 1, **RQ2**: **"Are human-written implementations more memory-efficient than those generated by Llama-3.3 and**

Table 4. Correctness Evaluation of Llama-3.3 Solutions for Rosetta Code Tasks

Problem	Prompt Met?	Executable Code?	Correct Output?	Procedural Logic?	Extra Text?	Clarity?	Omissions?
ABC problem	Yes	Yes	Yes	Yes	Yes	Good	No
Abundant deficient and perfect number classifications	Yes	Yes	Yes	Yes	No	Good	No
Abundant odd numbers	Yes	Yes	Partial	Yes	No	Good	Yes
Additive primes	Yes	Yes	Yes	Yes	No	Very Good	No
Almkvist–Guillera formula for pi	Yes	Yes	Yes	Yes	No	Good	No
Attractive numbers	Yes	Yes	Yes	Yes	No	Good	No
Countdown	Yes	Yes	Partial	Partial	No	Good	Yes
CORDIC	Yes	Yes	Partial	Partial	No	Regular	Yes
Cycle detection	Yes	Yes	Partial	Partial	No	Good	Yes
Dijkstra Algorithm	Yes	No	No	No	No	Regular	Yes
Digit fifth powers	Yes	Yes	Yes	Yes	No	Good	No
Discrete Fourier transform	Yes	Yes	Yes	Yes	No	Very Good	No
English cardinal anagrams	Yes	No	Partial	Partial	No	Regular	Yes
Exactly three adjacent 3 in lists	Yes	Yes	Yes	Yes	No	Good	No
Extended straddling checkerboard	Yes	Yes	Partial	Partial	No	Regular	Yes
Free polyominoes enumeration	Yes	No	No	Partial	No	Regular	Yes
100 doors	Yes	Yes	Partial	Yes	No	Very Good	Yes
Find the intersection of two lines	Yes	Yes	Yes	Yes	No	Very Good	No
Iterators	Partial	Yes	Partial	Partial	Yes (prints)	Good	Partial
Largest product in a grid	Yes	No	No	Partial	No	Good	Yes
Last Friday of each month	Yes	Yes	Yes	Yes	No	Very Good	No
Long stairs	Yes	No	Partial	Partial	No	Good	Yes
Magic numbers	Yes	Partial	No	Partial	Yes (prints)	Weak	Yes

Table 5. Correctness Evaluation of Gemma2 Solutions for Rosetta Code Tasks

Problem	Prompt Met?	Executable Code?	Correct Output?	Procedural Logic?	Extra Text?	Clarity?	Omissions?
ABC problem	Yes	Yes	Yes	Yes	No	Good	No
Abundant deficient and perfect number classifications	Yes	Yes	No	Partial	No	Good	Yes
Abundant odd numbers	Yes	Yes	Partial	Partial	No	Regular	Yes
Additive primes	Yes	Yes	Yes	Yes	No	Good	No
Almkvist–Guillera formula for pi	Yes	No	No	No	No	Regular	Yes
Attractive numbers	Yes	Yes	No	Partial	No	Regular	Yes
Countdown	Yes	Yes	No	Partial	No	Regular	Yes
CORDIC	Yes	Yes	No	No	No	Regular	Yes
Cycle detection	Yes	Yes	Yes	Yes	No	Good	No
Dijkstra algorithm	Yes	Yes	Partial	Partial	No	Good	Yes
Digit fifth powers	Yes	Yes	Yes	Yes	No	Good	No
Discrete Fourier transform	Yes	Yes	Partial	Partial	No	Good	Yes
English cardinal anagrams	Yes	Yes	No	No	No	Weak	Yes
Exactly three adjacent 3 in lists	Yes	Yes	Yes	Yes	No	Good	No
Extended straddling checkerboard	Yes	Yes	No	No	No	Weak	Yes
Free polyominoes enumeration	Yes	Yes	No	No	No	Weak	Yes
100 doors	Yes	Yes	Yes	Yes	No	Good	No
Find the intersection of two lines	Yes	No	No	No	No	Weak	Yes
Iterators	Partial	Yes	Partial	Partial	Yes (prints)	Regular	Yes
Largest product in a grid	Yes	No	No	Partial	No	Regular	Yes
Last Friday of each month	Yes	Yes	No	No	No	Good	Yes
Long stairs	Yes	Yes	No	No	Yes (prints)	Weak	Yes
Magic numbers	Yes	Yes	Partial	Partial	No	Good	Partial

Table 6. Percentage of Correct, Partially Correct, and Incorrect Codes

Model	Correct	Partially Correct	Incorrect
Llama-3.3	43% (10/23)	39% (9/23)	17% (4/23)
Gemma2	26% (6/23)	26% (6/23)	48% (11/23)

Gemma2?". To this end, the analysis utilizes the data obtained from the experiments, presented in Figure 1, which directly compares the memory consumption (in bytes) for each problem, considering separately the LLM solutions and the human references from Rosetta Code.

Figure 1 presents a bar chart in which each compared problem shows, side by side, the memory consumption of the Llama-3.3, Gemma2, and Rosetta Code implementations. There is a wide variation across problems regarding the absolute amount of memory utilized by each approach. In many of the problems, especially those involving algorithmic logic and mathematics (such as "ABC problem", "Additive primes", "100 doors", "Exactly three adjacent 3 in lists"), human implementations typically exhibit memory consumption equal to or lower than that of the LLM solutions. Furthermore, the null values observed in the results stem from incorrect LLM implementations that failed to execute. For example, Gemma2 produced an inexecutable solution for the 'Intersection of two lines' task, which resulted in an absence of measurable performance data. This case relates to one of the **Incorrect** code in Table 6.

On the other hand, there are situations where the consumption of the LLMs is close to or even higher than that of the human references, especially when they adopt more direct or poorly optimized solutions. In problems that require the use of complex data structures or intense manipulation of arrays, lists, and matrices (such as in "Largest product in a grid" or "Discrete Fourier transform"), these differences become even more evident: human codes typically resort to leaner strategies, reducing the use of temporary variables, preferring in-place iteration, or leveraging built-in language functions. In contrast, in the experiments with Llama-3.3 and Gemma2, it was common to observe redundant snippets, auxiliary buffers without real necessity, or variables kept beyond their useful lifetime, which ultimately increased memory consumption.

An illustrative example can be seen in the **100 doors** problem. The human-written implementation, taken from Rosetta Code, is remarkably lean, as shown in Listing 3.

Listing 3. Rosetta Code - 100 Doors

```
for i in range(1, 101):
  if i**0.5 % 1:
    state='closed'
  else:
    state='open'
  print("Door {}:{}".format(i, state))
```

In this listing, no additional data structure is maintained in memory: the verification of open/closed doors is performed in a purely mathematical way, exploiting the property that only doors in positions of perfect square numbers remain open after all passes. That is, for each door, only a small arithmetic operation is performed, followed by a print statement; once each iteration is finished, no space is reserved for arrays, lists, or auxiliary buffers, resulting in minimal memory consumption.

On the other hand, the implementation generated by Llama-3.3 opted for a more procedural approach, faithful to the literal description of the problem, yet considerably more costly in terms of memory usage, as seen in Listing 4.

Listing 4. Llama3.3 generated code - 100 Doors

```
def door_state(n):
  doors = [False] * (n + 1)
  for i in range(1, n + 1):
    for j in range(i, n + 1, i):
      doors[j] = not doors[j]
  return [i for i, x in enumerate(doors) if
      x and i != 0]

def main():
  n = 100
  result = door_state(n)
  print("Open doors:", result)

if __name__ == "__main__":
  main()
```

In this case, the algorithm explicitly creates a list named `doors` with $n+1$ boolean elements (in the example, 101 when $n = 100$), where index 0 is not used, and each position from 1 to 100 represents the state of a door throughout the entire process. In each iteration, the function traverses this list and toggles the door states according to the current step, storing everything in memory until the end of the execution. Only at the end are the indices corresponding to the open doors collected. This method, while faithful to the problem description and easy to understand, requires the permanent allocation of a data structure proportional to the number of analyzed doors (plus an extra unused element), resulting in slightly higher memory usage.

This difference clearly illustrates how a human programmer's experience allows for the recognition of opportunities for mathematically compact solutions, avoiding the creation of unnecessary objects or structures. Meanwhile, LLMs tend to prioritize clarity and generality, which, in this practical scenario, sacrifices memory economy.

In specific cases, Rosetta Code also benefits from the use of Python's built-in functions, whereas LLMs tend to implement again native functionalities (for example, using explicit loops for filtering or list processing), which can consume more memory by creating unnecessary intermediate objects. These differences are largely due to the experience and critical eye of human programmers, who can perceive resource limitations and apply practices focused on efficiency and conscious memory use. In the case of LLMs, what is frequently observed is functional but more generic code that is less attentive to performance details; after all, the priority is often to deliver a

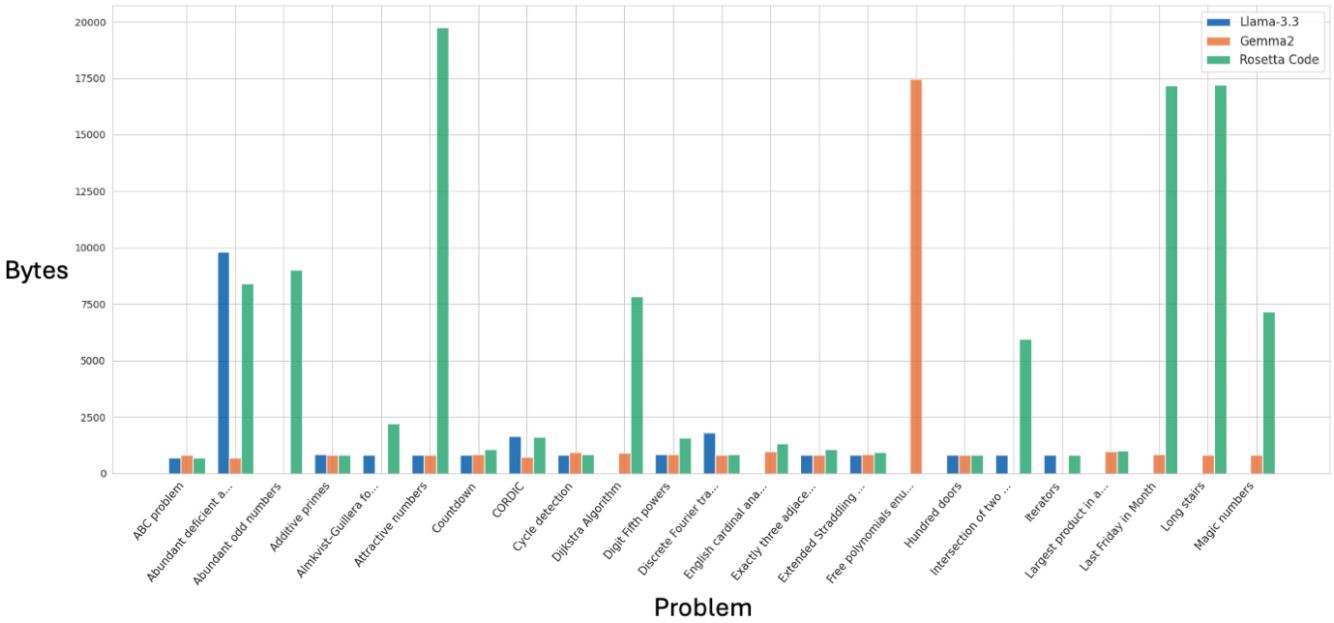

Figure 1. Memory Consumption Results

clear and complete response, even if it implies spending more resources.

With this, we can state that, in most of the tested problems, human-written implementations were more efficient in terms of memory consumption compared to the solutions generated by Llama-3.3 and Gemma2, despite occasional exceptions. These exceptions occur, for example, when the LLMs happen to generate leaner code or something similar to the human version, or in cases where the problem itself naturally restricts the amount of memory used. In general, the experience and attention of human programmers toward saving resources, even in small implementation details, result in lighter solutions. Thus, the answer to **(RQ2)** is affirmative: in most of the analyzed cases, human implementations demonstrated higher memory consumption efficiency than those generated by the Llama-3.3 and Gemma2 models.

3.3 CPU Usage

The CPU usage analysis, detailed in Figure 2, allows us to answer **RQ3: "Are human-written implementations more CPU-efficient than those generated by Llama 3 and Gemma 2"**. Based on the conducted experiments, it is possible to directly compare the approaches generated by the Llama-3.3 and Gemma2 models, as well as the reference human implementations from Rosetta Code, considering the percentage of CPU utilized during the execution of each task.

Figure 2 provides a view of the CPU percentage consumption for each evaluated Rosetta Code problem. It is noted that, in most problems, Llama-3.3 and Rosetta Code exhibit relatively close and moderate CPU usage, generally ranging between 5% and 10%. On the other hand, Gemma2 shows much more pronounced variations in several cases, with much higher consumption peaks. Analogously to Section 3.2, the null values represent the cases in which the LLM provides an incorrect implementation.

A notable difference when analyzing the figure is the extremely high CPU consumption presented by Gemma2 in the **Discrete Fourier Transform** task, exceeding 95% utilization, in contrast to the much lower values observed for both Llama-3.3 and the human reference from Rosetta Code (7.77% and 5.98%, respectively).

Upon analyzing the generated codes, it is noticeable that all of them follow the same line of a classic quadratic approach—meaning the execution time increases proportionally to the square of the input size. The difference lies in the details: each one brings its own structural choices and minor optimizations. Gemma2's code (Listing 5), for instance, stands out for being quite lean and easy to understand, but it ultimately falls short in some areas, unnecessarily repeating code segments and failing to handle common situations well. A clear case is the use of nested loops where it would be possible to vectorize operations or leverage more optimized libraries. This type of choice, when applied to large lists, causes both execution time and CPU consumption to skyrocket without bringing any real gain from a functional perspective.

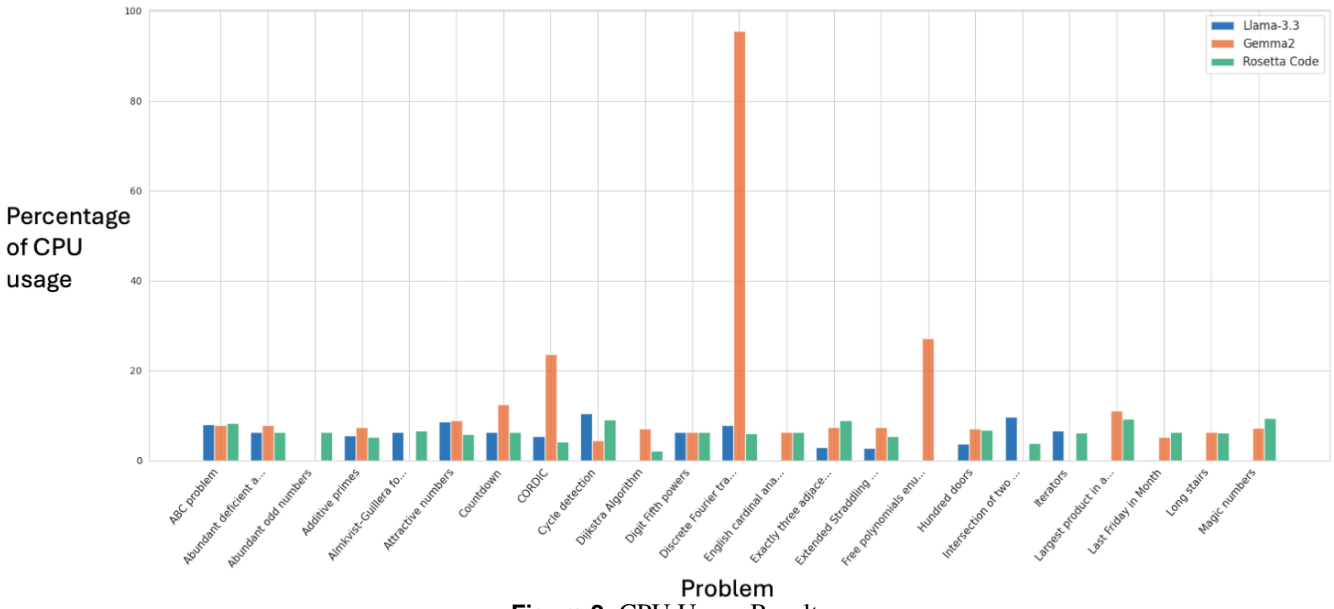

Figure 2. CPU Usage Results

Listing 5. Gemma2 generated code - Discrete Fourier Transform

```
def dft(x):
 N = len(x)
 X = np.zeros(N, dtype=complex)
 for n in range(N):
  for k in range(N):
   X[n] += x[k] * np.exp(-2j * np.pi * k *
       n / N)
 return X

def idft(X):
 N = len(X)
 x = np.zeros(N, dtype=complex)
 for n in range(N):
  for k in range(N):
   x[n] += X[k] * np.exp(2j * np.pi * k *
       n / N)
 return x / N
```

Another important point is that, unlike Llama-3.3 (Listing 6), some codes do not include basic precautions such as clearing numerical noise or checking boundary conditions. Llama-3.3 handles this well, using helper functions that maintain precision without putting a strain on the machine. Meanwhile, the human version from Rosetta Code stands out even more: in addition to being clear and well-commented, it features high portability, as it uses only standard library functions and features, avoiding resources that depend on specific platforms or versions. This facilitates code execution across different environments and systems—a typical concern of those who truly master the problem. The result is code that is efficient, resource-economical, and easy to understand.

Listing 6. Llama-3.3 generated code - Discrete Fourier Transform

```
def dft(x):
 N = len(x)
 X = np.zeros(N, dtype=complex)
 for n in range(N):
  for k in range(N):
   X[n] += x[k] * np.exp(-2 * np.pi * 1j * k * n / N)
 return X

def idft(X):
 N = len(X)
 x = np.zeros(N, dtype=complex)
 for n in range(N):
  for k in range(N):
   x[n] += X[k] * np.exp(2 * np.pi * 1j * k * n / N)
   x[n] /= N
 return x

def clean(x, threshold=1e-10):
 return np.where(np.abs(x) < threshold, 0, x)

def verify_dft():
 x = np.array([2, 3, 5, 7, 11])
 X = dft(x)
 x_reconstructed = idft(X)
 x_reconstructed = clean(x_reconstructed)
 return np.allclose(x, x_reconstructed)

def main():
 x = np.array([2, 3, 5, 7, 11])
 X = dft(x)
 x_reconstructed = idft(X)
 x_reconstructed = clean(x_reconstructed)
 print("Original sequence:", x)
 print("DFT:", X)
 print("Reconstructed sequence:", x_reconstructed)
 print("Verification:", verify_dft())

if __name__ == "__main__":
 main()
```

This example shows that it is not enough for code to be "simple" or to merely run without errors: if best practices and optimizations are lacking, performance drops and CPU usage skyrockets. In other words, the analysis of Figure 2 allows

us to answer **RQ3**: overall, human-made implementations manage to better balance clarity and efficiency, using fewer CPU resources than the solutions automatically generated by LLMs.

3.4 Execution Time

In this section, we address the fourth research question presented in Table 1, **RQ4**: **"Are human-written implementations more runtime-efficient than those generated by Llama-3.3 and Gemma2?"**. To address this question, we analyze the results synthesized in Figure 3, which presents a comparison of the execution time (in seconds, on a logarithmic scale) for each solved Rosetta Code problem.

As illustrated in Figure 3, execution times for the majority of problems are minimal and cluster near zero on the logarithmic scale. This indicates that both the LLMs and human developers implemented solutions efficiently, particularly for tasks with lower computational complexity or those where optimized algorithms were utilized. Unlike the metrics for memory consumption and CPU usage, we select a logarithmic scale for execution time to accommodate the significant variance in performance data. Utilizing a standard arithmetic scale in this context would have obscured the differences between faster implementations and diminished the overall legibility of the results.

However, there are cases where a discrepancy in execution times is observed. An example is the **Abundant deficient and perfect Number classification** problem. As shown in Figure 3, the Llama-3.3 implementation exhibits a high execution time (37.1 seconds), while the Rosetta Code implementation is also relatively slow (18.4 seconds). On the other hand, the Gemma2 code (Listing 7) solved the same problem almost instantaneously, with an execution time of fractions of a millisecond. The use of factorization up to the square root of n allows Gemma2 to drastically reduce the computational cost, justifying why its execution time, as shown in the orange bar, is substantially lower than that of the other solutions.

Listing 7. Gemma2 code - Abundant deficient and perfect number classifications

```
def classify_number(n):
 sum_divisors = 1
 for i in range(2, int(n**0.5) + 1):
  if n % i == 0:
   sum_divisors += i
   if i != n // i:
    sum_divisors += n // i
  if sum_divisors < n:
   return "deficient"
  elif sum_divisors == n:
   return "perfect"
  else:
   return "abundant"

deficient_count = 0
perfect_count = 0
abundant_count = 0
for i in range(1, 20001):
 if classify_number(i) == "deficient":
  deficient_count += 1
 elif classify_number(i) == "perfect":
  perfect_count += 1
 else:
  abundant_count += 1

print(f"Deficient: {deficient_count}")
print(f"Perfect: {perfect_count}")
print(f"Abundant: {abundant_count}")
```

This difference stems directly from the approaches adopted in the generated codes, each solving the challenge of identifying the quantity of abundant, deficient, and perfect numbers in the range from 1 to 20,000. For instance, in the Llama-3.3 code (Listing 8), for each number in the range, the sum of proper divisors is calculated in a rather naive manner, exhaustively iterating from 1 to $n-1$, which results in high computational complexity and elevated execution time.

Listing 8. Llama-3.3 code - Abundant deficient and perfect number classifications

```
def classify_numbers(n):
 def sum_of_proper_divisors(num):
  return sum(i for i in range(1, num) if num % i == 0)

  deficient_count = 0
  perfect_count = 0
  abundant_count = 0

  for i in range(1, n + 1):
   sum_of_divisors = sum_of_proper_divisors(i)
   if sum_of_divisors < i:
    deficient_count += 1
   elif sum_of_divisors == i:
    perfect_count += 1
   else:
    abundant_count += 1
  return deficient_count, perfect_count, abundant_count

def main():
 n = 20000
 deficient, perfect, abundant = classify_numbers(n)
 print(f"Deficient: {deficient}, Perfect: {perfect},
       Abundant: {abundant}")

main()
```

Therefore, the answer to **RQ4** is that time efficiency is not necessarily linked to whether the implementation is human or automated, but depends primarily on the algorithmic strategy employed to solve the problem. In other cases shown in

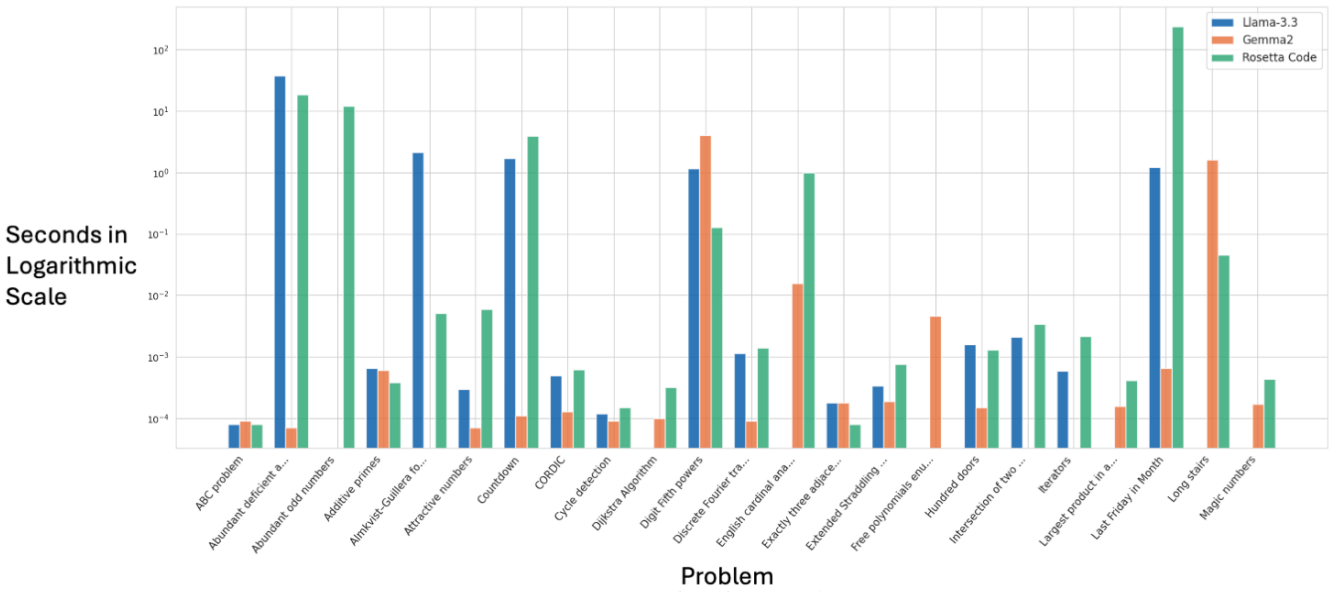

Figure 3. Execution time results

Figure 3, the solutions presented similar execution times; it is important to note that null values indicate problems that were not executed for a given model, rather than negligible times.

A detailed analysis of the graphs reinforces the importance of the chosen resolution method and shows that both humans and LLMs can generate implementations that are either fast or slow, depending on the chosen strategies.

3.5 Discussion

The joint evaluation of the results allows us to see a more complete picture of what LLMs truly offer in code generation—their strengths, limitations, and potential. This perspective does not rely solely on isolated metrics, but on how they relate to one another: correctness **(RQ1)**, memory efficiency **(RQ2)**, CPU usage **(RQ3)**, and execution time **(RQ4)**.

Analyzing these in an integrated manner, it is observed that efficiency factors (**RQ2, RQ3, RQ4**) are often interconnected: codes that consume more memory, for instance, typically exhibit higher execution times and CPU consumption, suggesting that choices in algorithm structure and modeling impact multiple performance indicators. The extreme case of **Discrete Fourier Transform** for the Gemma2 model illustrates this well: the code simultaneously generated high CPU usage and execution time, stemming from a lack of optimization and the repetition of unnecessary operations. Analogous situations can be observed in other problems, such as "**100 doors**".

However, there are revealing and intriguing exceptions. In mathematically simple problems, or those whose resolution is naturally resource-constrained (e.g., "**Find the intersection of Two Lines**" or "**Attractive Numbers**"), LLMs—especially when they hit upon an optimized approach—can outperform human performance, including cases with 13 times lower memory consumption or 219 times faster execution. These episodes indicate that, despite being inconsistent, LLMs have the potential to generate highly efficient solutions, rivaling or even exceeding traditional routines, particularly when they fully exploit the programming language and its native libraries.

Regarding **RQ1**, it is worth noting that errors made by LLMs are rarely merely syntactic; failures in understanding the problem description, omissions of edge cases, or procedural logic confusion predominate, confirming the critical need for human review. In cases where such issues do not occur, script execution tends to rival the best solutions from Rosetta Code.

Finally, it is also important to discuss strong numerical claims. Table 7 summarizes the most significant performance disparities observed during the experiments. In terms of memory consumption, the efficiency gains observed for the **Abundant deficient and perfect number classifications** and **Attractive numbers** tasks (11.91x and 24.26x, respectively) stem from Gemmas2 ability to generate scripts without the auxiliary buffers or redundant data structures present in the human Rosetta Code. The most notable finding is the execution time disparity in the **Abundant deficient and perfect number classifications** task, where a performance gain exceeding 260,000x was recorded. This difference is directly traceable to the underlying algorithmic strategy: while the human implementation utilizes a naive exhaustive iteration, the LLM adopts a more efficient $O(\sqrt{n})$ factorization approach. Conversely, the results for the **Discrete Fourier transform** reveal a significant CPU regression, with the automated solution incurring a 15.96x increase in resource overhead due to a lack of vectorization and the reproduction of unoptimized loop structures.

Table 7. Traceability of Strong Performance Claims (Gemma2 vs. Human)

Metric	Rosetta Code Task	Human (Ref)	Gemma2	Calculated Ratio
Memory Consumption	Abundant, Def., Perfect Class.	8,390 bytes	704.32 bytes	11.91x
Memory Consumption	Attractive Numbers	19,736.72 bytes	813.50 bytes	24.26x
Execution Time	Abundant, Def., Perfect Class.	18.39 s	0.00007 s	262,714x
CPU Usage	Discrete Fourier Trans.	5.98%	95.49%	-15.96x

3.6 Statistical analysis

This section presents statistical analyses for our data. Section 3.6.1 presents the Wilcoxon Signed-Rank Test [21] results while Section 3.6.2 discusses the the Vargha-Delaney's A Effect Size [22] analysis for non-parametrical test and effect size measures, respectively.

3.6.1 Wilcoxon Signed-Rank test

Table 8 illustrates the Wilcoxon Signed-Rank Test [21] results for memory consumption, CPU usage, and execution time. This test reveals significant differences in memory consumption patterns across Llama3, Gemma2, and Rosetta Code (Human). Notably, the comparison between Gemma2 and Human shows highly significant differences (p = 0.0005), with Human consuming more memory (mean: 10,889.71 bytes) compared to Gemma2 (mean: 840.15 bytes), representing approximately a 13-fold increase. In contrast, the Llama3 vs. Gemma2 comparison shows no significant difference (p = 0.9165), despite Llama3 having a higher mean (1,532.89 bytes), as both models exhibited similar median values around 828 bytes. The Llama3 vs. Human comparison is marginally non-significant (p = 0.0843), suggesting that while Human tends to allocate more memory, this difference approaches statistical significance and could manifest with larger sample sizes. These findings indicate that Gemma2 and Llama3 employ memory-efficient implementations, whereas Human's approach requires substantially more memory allocation for equivalent computational tasks.

Additionally, CPU usage illustrated in Table 8 demonstrates distinct patterns in computational efficiency and parallelization strategies among the two models and Human. Llama3 vs. Gemma2 comparison reveals significant differences (p = 0.0131), with Gemma2 consuming substantially more CPU resources (mean: 15.29%) compared to Llama3 (mean: 5.73%), indicating that Gemma2 employs more aggressive parallelization and multi-threaded execution strategies. Conversely, the Llama3 vs. Human comparison shows no significant differences in CPU usage (p = 0.5509), with both models maintaining similar mean CPU consumption around 6%, suggesting comparable computational efficiency in their sequential processing approaches. The Gemma2 vs. Human comparison also reveals significant differences (p = 0.0232), with Gemma2 demonstrating higher CPU utilization (mean: 13.79% vs. 6.38%), reinforcing the observation that Gemma2 leverages parallel computation more effectively. These results suggest that while Gemma2 achieves superior performance through intensive CPU parallelization, Llama3 and Human maintain more conservative resource consumption patterns, albeit with different implications for execution speed.

Moreover, execution time comparisons reveal the most dramatic performance disparities between Llam3, Gemma2, and Human. The Llama3 vs. Gemma2 comparison shows significant differences (p = 0.0231), with Gemma2 demonstrating superior speed (median: 0.00014s) compared to Llama3 (median: 0.000625s), indicating approximately 4.5 times faster execution. The Llama3 vs. Human comparison shows no significant differences in execution time (p = 0.5509), with both models exhibiting similar median values around 0.001s, despite notable differences in their mean values driven by occasional computational outliers.

As detailed in Section 3.5, the Gemma2 vs. Human comparison yielded statistical significance ($p = 0.0448$) and a large inverse effect ($A = 0.2526$). These findings collectively demonstrate that Gemma2 achieves the most efficient execution profile, Llama3 provides intermediate performance with consistent reliability, and Human exhibits the most substantial execution overhead, particularly for complex computational tasks.

In summary, Gemma2 demonstrates the most efficient resource utilization across memory consumption, CPU usage, and execution time. On the other hand, Llama3 provides balanced performance with minimal resource overhead while Human exhibits the highest resource consumption, particularly in memory and execution time. Finally, statistical significance is most pronounced in Gemma2 vs. Human comparisons, indicating fundamental differences in implementation strategies.

3.6.2 Vargha-Delaney's A Effect Size

Table 9 illustrates the results for the Vargha-Delaney's A Effect Size [22] analysis. Regarding memory consumption, it reveals pronounced disparities between Llama3, Gemma2, and Rosetta Code (Human), with effect sizes that escalate dramatically across comparisons. The comparison between Llama3 and Gemma2 shows a negligible effect (A = 0.5612), which indicates that both models employ similar memory allocation strategies with comparable median values around 828 bytes, despite Llama3's higher mean driven by occasional outliers. In contrast, the Llama3 versus Human comparison demonstrates a medium inverse effect (A = 0.2822), suggesting Human has approximately 72% probability of exceeding Llama3 in memory consumption. Most critically, the Gemma2 versus Human comparison reveals an overwhelming effect (A = 0.1574), indicating that Human has 84% probability of consuming significantly more memory, with a striking 13-fold difference in mean values (10,890 bytes for Human versus 840

Table 8. Wilcoxon Signed-Rank Test Results: Pairwise Comparisons of Memory, CPU, and Execution Time

Comparison	Metric	N	Mean 1	Mean 2	Median 1	Median 2	V-Statistic	P-value
Llama3 vs Gemma2	Memory (bytes)	14	1532.89	832.69	828.04	823.39	44.00	0.9165
	CPU (%)	14	5.73	15.29	6.25	7.58	10.00	0.0131*
	Time (s)	14	2.941	0.290	0.000625	0.000140	13.00	0.0231*
Llama3 vs Human	Memory (bytes)	15	1486.66	3597.53	831.44	1072.32	25.00	0.0843
	CPU (%)	15	5.70	6.53	6.25	6.24	43.00	0.5509
	Time (s)	15	2.888	1.494	0.001130	0.001290	43.00	0.5509
Gemma2 vs Human	Memory (bytes)	17	840.15	10889.71	828.32	1340.70	9.00	< 0.001**
	CPU (%)	17	13.79	6.38	7.41	6.24	29.00	0.0232*
	Time (s)	17	0.336	15.211	0.000160	0.001290	34.00	0.0448*

*p ¡ 0.05 (significant at $\alpha = 0.05$)

**p ¡ 0.001 (highly significant)

bytes for Gemma2). This very large inverse effect, combined with extreme statistical significance (p ¡ 0.001), provides conclusive evidence that Gemma2 implements substantially more memory-efficient algorithms or data structures.

Table 9. Memory Consumption: Vargha-Delaney's A Effect Sizes

Comparison	N	Mean 1	Mean 2	A Value	Effect Magnitude
Llama3 vs Gemma2	14	1532.89	832.69	0.5612	Small
Llama3 vs Human	15	1486.66	3597.53	0.2822	Medium (inv)
Gemma2 vs Human	17	840.15	10889.71	0.1574	Large (inv)

Additionally, the Vargha-Delaney's A analysis of CPU utilization patterns reveals distinctly different computational strategies employed by each model and human implementations. Table 10 illustrates the results. The comparison between Llama3 and Gemma2 demonstrates a large inverse effect (A = 0.2270), which indicates that Gemma2 has 77% probability of consuming higher CPU resources (mean: 15.29% versus 5.73%). This represents a tradeoff wherein Gemma2 sacrifices CPU efficiency to achieve superior overall performance through multi-threading and parallel execution. The Llama3 versus Human comparison shows a negligible effect (A = 0.4400), revealing nearly equivalent CPU utilization patterns with both approaches operating at approximately 6% CPU. This suggests comparable sequential processing efficiency and minimal reliance on parallelization strategies. Finally, the Gemma2 versus Human comparison exhibits a large direct effect (A = 0.7647), confirming Gemma2's 76% probability of higher CPU usage relative to Human. These effect sizes collectively demonstrate that Gemma2's exceptional speed and memory efficiency are fundamentally achieved through CPU-intensive parallel processing, representing a deliberate architectural choice to maximize computational throughput by leveraging available CPU capacity. On the other hand, Llama3 and Human maintain more conservative CPU consumption patterns, with Gemma2's elevated CPU usage representing an optimal and justified resource allocation strategy for time-critical applications.

Furthermore, Table 11 illustrates the Vargha-Delaney's A analysis of execution time, which demonstrates the most dramatic performance disparities among all metrics examined, with effect sizes that unequivocally establish Gemma2's computational superiority. The comparison between Llama3 and Gemma2 reveals a large effect (A = 0.8087), indicating Llama3 has 81% probability of exceeding Gemma2 in execution duration, with median values demonstrating Gemma2's approximately 4.5-fold speed advantage (0.00014 seconds versus 0.000625 seconds). This substantial effect size validates the statistical significance observed in Wilcoxon testing (p = 0.0231) presented in Section 3.6.1. It also indicates Gemma2's efficiency gains are both statistically meaningful. The Llama3 versus Human comparison demonstrates a negligible effect (A = 0.5111). This reveals essentially equivalent execution performance at the median level (approximately 0.001 seconds for both), despite notable mean differences driven by computational outliers. The near-0.5 effect size indicates that Llama3 and Human exhibit fundamentally similar algorithmic efficiency for typical tasks, suggesting comparable underlying implementations with equivalent processing logic.

Table 10. CPU Utilization: Vargha-Delaney's A Effect Sizes

Comparison	N	Mean 1 (%)	Mean 2 (%)	A Value	Effect Magnitude
Llama3 vs Gemma2	14	5.73	15.29	0.2270	Large (inv)
Llama3 vs Human	15	5.70	6.53	0.4400	Small (inv)
Gemma2 vs Human	17	13.79	6.38	0.7647	Large (direct)

Also, as showed in the Section 3.5, the Gemma2 vs. Human comparison yielded statistical significance ($p = 0.0448$) and a large inverse effect ($A = 0.2526$). The collective pattern of execution time effects, with Gemma2 demonstrating decisive speed advantages (A = 0.8087 versus Llama3, A = 0.2526 versus Human) while Llama3 and Human show negligible differentiation (A = 0.5111), states that Gemma2 is superior for latency-sensitive applications.

Table 11. Execution Time: Vargha-Delaney's A Effect Sizes

Comparison	N	Mean 1 (s)	Mean 2 (s)	A Value	Effect Magnitude
Llama3 vs Gemma2	14	2.941	0.290	0.8087	Large
Llama3 vs Human	15	2.888	1.494	0.5111	Negligible
Gemma2 vs Human	17	0.336	15.211	0.2526	Large (inv)

Last but not least, Gemma2 dominates in memory efficiency (A = 0.1574) and speed (A = 0.2526 vs Human; A = 0.8087 vs Llama3) while Higher CPU usage (A = 0.2270, A = 0.7647) is justified by performance gains. Moreover, Llama3

is a balanced performer with small-to-medium effects vs both alternatives. There is a negligible difference from Human in time (A = 0.5111) and CPU (A = 0.4400), but better memory profile (A = 0.2822). At last, human code is inferior across critical metrics with large inverse effects indicating lower measures in both memory and time.

3.7 Threats to Validity

This section discusses threats to the validity of our work.

External Validity. The evaluation relies on a sample of 23 tasks extracted from the Rosetta Code platform, categorized into Mathematics and Number Theory, Algorithmic and Logic Puzzles, and data and string processing. While these tasks represent well-defined computational problems whose solutions are attainable through computation, they are inherently self-contained challenges and may not fully reflect the structural complexities of large-scale, real-world industrial software development.

Furthermore, the analysis was exclusively conducted using the Python programming language. Because different programming languages have distinct memory management mechanisms and execution profiles, the efficiency and resource consumption patterns observed might not automatically generalize to low-level languages or different programming paradigms.

Another external validity threat concerns the specific Large Language Models (LLMs) selected for the experiment. This work specifically investigates the correctness and efficiency of codes generated by Llama-3.3-70B Versatile and Gemma2-9B Instruct. Therefore, the findings regarding memory consumption, CPU usage, and execution time are strictly bound to the specific transformer architectures, training pipelines, and parameter sizes of these two models. Consequently, the performance limitations or algorithmic inefficiencies observed might not mirror the behavior of other proprietary models or newly released open-weight alternatives.

Internal Validity. First, the construction of the prompts and the interaction with the model APIs could influence the generated outcomes. The prompt template was carefully structured into three main parts—context, problem description, and expected task—and was written entirely in English to leverage the models' native optimization. However, alternative prompting techniques, such as few-shot prompting or chain-of-thought, could potentially guide the models to yield more optimized algorithmic logic. Additionally, while the API requests to Groq Cloud were configured with a temperature of 0 to minimize randomness in the output, inherent non-determinism in the underlying accelerated computing hardware or minor API updates during the testing phase could theoretically introduce slight variations in the generated responses.

A second threat to internal validity relates to the environment and procedures used for the computational measurements. To mitigate random fluctuations in system performance, the measurement script was designed to execute each task implementation one hundred times in a loop, extracting the average values for memory, execution time, and CPU utilization. All measurements were conducted in a controlled environment running the Ubuntu operating system with an AMD Ryzen 7-5800H processor and 32 GB of RAM. Despite these precautions, background processes intrinsic to the operating system or the automatic garbage collection mechanisms native to Python could still insert residual noise into the performance metrics collected during the snapshot windows.

Construct Validity. It involves the metrics used to evaluate the implementations and the baselines chosen for comparison. The study uses human-written implementations available on Rosetta Code as the primary reference for correctness and efficiency. While these solutions provide a practical baseline backed by the collaborative nature of the platform, they are not strictly guaranteed to be the absolute most computationally efficient Python codes mathematically possible. Furthermore, the Percentage of Correct Implementations (`PCI`) metric requires evaluating whether the generated code is syntactically correct and presents the expected logic to solve the proposed problem. The qualitative assessment of dimensions such as procedural logic, output accuracy, and solution clarity involves manual analysis, which inherently introduces a degree of human judgment into the correctness evaluation.

4. Related Work

In this section, relevant related works are discussed, highlighting their similarities and differences compared to the present study, as well as the specific contributions of each approach to the evaluation of code performance and correctness.

The study by Pereira et al. [9] investigates the impact of different programming languages on energy consumption, execution time, and memory usage. The authors conduct experiments with various programming problems and compare languages such as C, Java, and Python, analyzing how the characteristics of each language affect energy and computational efficiency across different algorithms. The main conclusion of this work is that low-level languages, such as C, generally present better performance in terms of memory and time, while high-level languages, such as Python, offer greater ease of use and productivity, even at the cost of some efficiency.

While there is a relevant similarity regarding the interest in computational resource consumption, a significant difference compared to our work is that Pereira et al. [9] focus their analysis on the effect of programming language choice, including the issue of energy efficiency—an aspect not addressed in this research. In contrast, the study presented here focuses on the analysis of solutions generated by Large Language Models (LLMs), such as Llama-3.3 and Gemma2, evaluating their ability to produce correct and efficient Python scripts regarding memory usage, CPU, and execution time, in addition to comparing these results with human implementations available in Rosetta Code. Thus, while the related work em-

phasizes the choice of language as a central variable, our study investigates the potential and limitations of automatic code generation by LLMs, considering algorithms/functions within the same language (Python).

Another relevant work is by Liu et al. [7], which examines the functionality and correctness of code produced by generative language models, such as GPT-3.5 Turbo. To this end, the authors propose the EvalPlus framework, which enhances traditional code evaluation benchmarks, such as HumanEval [23], by significantly expanding the volume and variety of automated test cases. This allows for a more rigorous evaluation of the accuracy of algorithms produced by LLMs, showing that results obtained using only conventional benchmarks tend to overestimate the models' real performance.

Although both the work of Liu et al. [7] and the present thesis seek to evaluate the correctness of code generated by language models, the methodological approach is distinct. Liu et al.'s study primarily uses automated benchmarks and a large set of tests to assess functional accuracy. In our research, the analysis was conducted in a detailed and comparative manner, using human solutions from Rosetta Code as a reference and evaluating not only functional correctness but also aspects of computational performance (memory, CPU, time), which enables a broader understanding of the impact of automatic generation compared to human development.

In the work of Austin et al. [6], the ability of LLMs to generate scripts from natural language prompts is analyzed through specific benchmarks, such as Mostly Basic Programming Problems (MBPP) and MathQA-Python [24]. The study also incorporates an analysis of the impact of human feedback on improving generated code, promoting iterative adjustments that contribute to an increase in the models' success rate.

Similar to the current study, Austin et al. aim primarily to evaluate the models' aptitude for generating correct code from textual descriptions. However, beyond also relying almost exclusively on automated benchmarks, there is the distinction of continuous human feedback to the model for algorithm refinement—an approach not adopted in this work. In contrast, the present study opted to compare, under controlled and real conditions, model output with established human solutions, focusing not only on criteria of correctness and clarity but also on practical computational performance metrics.

Thus, it is observed that although they share the objective of evaluating the quality of code generated by LLMs, the related works discussed here differ from the present work in terms of methodology, scope of analysis, evaluation criteria considered (including practical performance rather than just functionality), or the use of human feedback. The main contribution of this research lies precisely in the direct and detailed comparison between automatically generated code and human solutions available in a public repository, using both classic metrics and computational performance aspects, which broadens the understanding of the real limitations and potentials of these technologies for automatic programming.

5. Conclusion

This research explored the intersection of automated code generation and digital sustainability, evaluating the computational efficiency of Large Language Models (LLMs) compared to human implementations. By analyzing the Llama-3.3 and Gemma2 models against human benchmarks from Rosetta Code, the work aimed to determine whether the rapid productivity gains provided by LLMs come at the expense of memory, CPU, and execution time efficiency. Ultimately, the goal was to provide an understanding of the performance trade-offs inherent in the transition toward LLM-assisted software development.

Regarding correctness, the study demonstrates that while LLMs have reached a level of maturity that allows for the generation of functional scripts for routine tasks, they still lack the critical nuance required for complex algorithmic optimization. The failures observed were rarely syntactic. They were predominantly rooted in procedural logic confusion and the omission of critical edge cases. This suggests that the current value of LLMs in programming lies in rapid prototyping rather than the generation of production-ready code, reinforcing the necessity of human-in-the-loop validation for ensuring logical soundness.

The broader takeaway on memory consumption reveals a fundamental divergence in implementation priorities. While human programmers leverage experience to minimize data structure overhead, LLMs tend to prioritize code clarity and a literal interpretation of problem descriptions. This often leads to an issue where automated solutions maintain unnecessary auxiliary buffers and redundant variables that increase the memory footprint compared to the leaner strategies utilized in human-written code. Consequently, memory efficiency of LLMs remains a distinct human advantage, driven by the ability to apply mathematical shortcuts and conscious resource management.

Additionally, CPU usage results indicate that the apparent simplicity of LLM-generated code can be deceptive from a resource perspective. Without the application of specific best practices, automated solutions frequently incur significant processing overhead. This finding highlights a critical risk for digital sustainability: the adoption of generic, automated logic in large-scale systems may lead to an increase in resource utilization that could increase operational costs.

In terms of execution time, the results emphasize that performance is primarily a function of algorithmic strategy rather than the origin of the source code. Both humans and LLMs are capable of producing either fast or slow implementations depending on the chosen resolution method. This study identified instances where the models' ability to synthesize an optimized mathematical approach allowed them to significantly outperform traditional iterative techniques, suggesting that LLMs can occasionally achieve efficiency gains that exceed the naive logic found in some human reference scripts.

Furthermore, the statistical analysis and effect size mea-

sures provide an additional validation of these findings, confirming that the performance disparities observed are not mere anomalies. The data reveals a clear trade-off: the achievement of superior speed and memory efficiency in automated models often requires an allocation of CPU resources. This quantifiable interaction between metrics suggests that performance in LLM-generated code is a multidimensional balance, where gains in one area typically necessitate a deliberate sacrifice in another.

In conclusion, the responsible adoption of LLMs in software engineering requires a shift from purely functional evaluations toward a performance-driven culture. While these tools offer a path toward increased developer productivity, true digital sustainability can only be achieved by integrating LLM-generated code with human expertise in algorithmic optimization. Ultimately, LLMs should be viewed as high-speed collaborative partners that require careful technical oversight to ensure that the code they produce is as computationally efficient as it is functional.

5.1 Future Work

The results of this work pave the way for a series of developments and enhancements in future research. One of the most immediate possibilities consists of expanding the set of evaluated language models, incorporating recent or alternative LLMs of different sizes and architectures to analyze in greater depth how model scale and training profiles influence the generation of efficient and correct code. Another important evolution would be to expand the scope of programming languages analyzed, including, in addition to Python, languages with distinct paradigms such as C, Java, and JavaScript, allowing for an investigation into differences in LLM performance across varied syntactic contexts and efficiency demands.

Furthermore, it is possible to increase the variety of evaluated tasks as well as consider other reference platforms, such as The Algorithms [25], LeetCode [26], Codeforces [27], and Beecrowd [28], providing an even more diversified base in terms of both the nature of the challenges and the quality of the comparing human solutions. A productive alternative would be to evaluate the models' robustness regarding the complexity and size of the problems, verifying to what extent they can generalize to challenges that go beyond the simple or routine examples found in traditional benchmarks.

Future investigations could also explore the impact of human feedback on the refinement process of the generated code, testing iterative and collaborative approaches between language models and experienced developers, which could contribute to raising correctness and efficiency rates to even higher levels.

Therefore, it is considered strategic to develop automated review or recommendation systems that can operate in conjunction with LLMs, mitigating potential deficiencies and steering automated solutions toward the quality standards required in productive and critical environments. These possible lines of continuity not only reinforce the relevance of the topic but also signal the vast field to be explored at the interface between Artificial Intelligence and Software Engineering.

Artifacts Availability

We provide all artifacts used in this study in our Online Appendix [29].

Author contributions

Jackson Lima: Conceptualization, Methodology, Software, Investigation, Data Curation, Writing – Original Draft. Rodrigo Andrade: Conceptualization, Methodology, Writing – Original Draft, Writing – Review and Editing. All authors have read and agreed to the submitted version of the manuscript.

References

[1] MASTROPAOLO, A. et al. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438*, 2023. Available at: ⟨https://arxiv.org/abs/2302.00438⟩.

[2] OPENAI. *ChatGPT*. 2024. Available at: ⟨https://openai.com/index/chatgpt/⟩.

[3] GOOGLE. *Gemini*. 2024. Available at: ⟨https://deepmind.google/technologies/gemini/⟩.

[4] Meta AI. *Llama 3*. 2024. Available at: ⟨https://llama.meta.com/⟩.

[5] ANTHROPIC. *Claude*. 2024. ⟨https://www.anthropic.com/claude/⟩.

[6] AUSTIN, J. et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. Available at: ⟨https://arxiv.org/abs/2108.07732⟩.

[7] LIU, J. et al. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2023. Available at: https://arxiv.org/abs/2305.01210.

[8] VARTZIOTIS, D.; MITROPOULOS, D. Green ai for code generation: Investigating the energy cost of large language models. *arXiv preprint arXiv:2402.12513*, 2024.

[9] PEREIRA, R. et al. Energy efficiency across programming languages: how do energy, time, and memory relate? In: *ACM SIGPLAN International Conference on Software Language Engineering*. [S.l.: s.n.], 2017.

[10] POP, M. *Measuring the energy consumption and carbon footprint of encrypted databases using CodeCarbon*. Tese (Doutorado) — University of Twente, 2025.

[11] STEENHOEK, B. et al. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218*, 2024. Available at: ⟨https://arxiv.org/html/2403.17218v1⟩.

[12] Rosetta Code. *Rosetta Code*. 2025. Available at: ⟨https://rosettacode.org/wiki/Rosetta_Code⟩.

[13] SALEWSKI, L. et al. In-context impersonation reveals large language models' strengths and biases. In: *Conference on Neural Information Processing Systems*. [S.l.: s.n.], 2023. Available at: ⟨https://arxiv.org/abs/2305.14930⟩.

[14] Meta AI, via GroqCloud. *Llama 3.3-70B Versatile [API documentation]*. 2024. Available at: ⟨https://console.groq.com/docs/model/llama-3.3-70b-versatile/⟩.

[15] Google DeepMind. *Gemma2-9B IT [API documentation]*. 2024. Available at: ⟨https://ai.google.dev/gemma/docs/⟩.

[16] Meta AI. *Llama 3*. 2024. ⟨https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3⟩.

[17] TOUVRON, H. et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. Available at: ⟨https://arxiv.org/abs/2307.09288⟩.

[18] OUYANG, L. et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, v. 35, p. 27730–27744, 2022.

[19] Google. *Gemma 2*. 2024. ⟨https://ai.google.dev/gemma/docs⟩.

[20] Groq, Inc. *GroqCloud API Documentation*. 2024. Available at: ⟨https://console.groq.com/docs/⟩.

[21] WILCOXON, F. Individual comparisons by ranking methods. *Biometrics bulletin*, v. 1, n. 6, p. 80–83, 1945.

[22] VARGHA, A.; DELANEY, H. D. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, v. 25, n. 2, p. 101–132, 2000.

[23] CHEN, M. et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. Available at: ⟨https://arxiv.org/abs/2107.03374⟩.

[24] AMINI, A. et al. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *arXiv preprint arXiv:1905.13319*, 2019. Available at: https://arxiv.org/abs/1905.13319.

[25] The Algorithms. *The Algorithms*. 2025. Available at: ⟨https://the-algorithms.com/⟩.

[26] LeetCode. *LeetCode Online Judge*. 2025. Available at: ⟨https://leetcode.com/⟩.

[27] Codeforces. *Codeforces: Competitive Programming Platform*. 2025. Available at: ⟨https://codeforces.com/⟩.

[28] Beecrowd. *Beecrowd: Programming Problems and Contests*. 2025. Available at: ⟨https://www.beecrowd.com.br/⟩.

[29] LIMA, J.; ANDRADE, R. *Analyzing Correctness, Memory Consumption, CPU Usage, and Execution Time of the LLMs Llama-3.3-70B Versatile and Gemma2-9B Instruct*. 2026. Available at: ⟨https://zenodo.org/records/18776535⟩.