Comparative of source code generated by principal LLM generators for Python and Lua languages

Gustavo Rossoni Correa de Barros, Mirkos O. Martins
Computer Science
UFN - Universidade Franciscana
Santa Maria - RS
gustavo.rossoni@ufn.edu.br, mirkos@ufn.edu.br

***Abstract*—As artificial intelligence becomes increasingly capable of tasks traditionally done by humans, its role in software development, particularly in code generation, is growing. This study evaluates the performance of code generation tools like GitHub Copilot, Meta CodeLLaMa 34b instruct, and ChatGPT 3.5 in generating Python and Lua code for software development tasks. Test-Driven Development (TDD) was employed as an analytical framework to assess accuracy, readability, performance, and complexity of the generated code. The findings reveal that while AI models show promise in code generation, their effectiveness varies significantly across different tasks and languages, highlighting the need for careful selection and validation when integrating AI tools into software development workflows.**

Keywords: **Large Language Models; Test-Driven Development; Code Analysis.**

I. Introduction

Software developers have raised concerns about the emergence of generative artificial intelligence tools that can generate source code as good as humans. This phenomenon has sparked discussions surrounding ethical implications, copyright infringement risks, and potential malicious consequences of unregulated usage [1].

Developers are particularly concerned about building trust in AI-powered code generation tools, as their adoption depends on understanding the appropriate levels of trust and how to design interfaces to facilitate it [2].

These challenges are further compounded by the fact that generative AI models, such as ChatGPT, have the potential to create serious online risks, including privacy and safety concerns [3].

In light of these issues, it becomes crucial to evaluate the performance, limitations, and applicability of AI-powered code generation tools in real-world software development contexts. This study aims to analyze the quality of code generated by large language models (LLMs) in Python and Lua languages using GitHub Copilot [4], Meta LLaMa 2 [1] [5], Gemini (previously known as Bard), Claude V2 from Anthropic [6], the Perplexity AI LLM-Powered search engine, and ChatGPT 3.5 [7] through Test-Driven Development (TDD) methodology [8].

[1]Specifically CodeLLaMa 34b instruct

The research objectives of this study are threefold: Firstly, to compare the code generation performance of different LLMs for Python and Lua programming languages. Secondly, to assess the readability, performance, and complexity of the generated code. Finally, to identify the strengths and weaknesses of each model in handling specific programming tasks.

II. Related Work

There has been a growing body of research focusing on the performance evaluation of LLMs in coding related tasks. Yuan et al. [9] evaluated 10 open-source instructed LLMs on four representative code comprehension and generation tasks. They found that for the zero-shot setting, instructed LLMs are very competitive on code comprehension and generation tasks and sometimes even better than small state-of-the-art models specifically fine-tuned on each downstream task. They also found that larger instructed LLMs are not always better on code-related tasks.

Liu et al. [10] performed a systematic empirical assessment of code generation using ChatGPT, a recent and popular LLM. Their evaluation encompassed a comprehensive analysis of code snippets generated by ChatGPT, focusing on three critical aspects: correctness, understandability, and security. They also specifically investigated ChatGPT's ability to engage in multi-round process (i.e., ChatGPT's dialog ability) of facilitating code generation.

Zhong et al. [11] studied the reliability and robustness of the code generation from LLMs. They proposed a dataset RobustAPI for evaluating the reliability and robustness of code generated by LLMs. They collected 1208 coding questions from StackOverflow on 24 representative Java APIs and evaluated them on current popular LLMs. The evaluation results showed that even for GPT-4, 62% of the generated code contains API misuses, which would cause unexpected consequences if the code is introduced into real-world software.

These studies provide valuable insights into the performance of LLMs in tackling code generation tasks and uncover potential issues and limitations that arise in the LLM-based code generation. They lay the groundwork for improving AI and LLM-based code generation techniques.

This article has an approach to measure different LLM's with a method that uses scores for accuracy, readability, complexity and performance in different categories of algorithms generated in two languages: Python and Lua.

III. Background

A. Large Language Models (LLMs)

A large language model (LLM) is a type of neural network architecture that is designed to predict the next word in a sequence based on its history. These models use deep learning techniques, specifically recurrent neural networks (RNNs) or transformer architectures, to learn the probability distribution over sequences of words. LLMs can be trained on massive amounts of text data to produce human-like outputs when generating new text [12] [13].

B. Python and Lua

Python: is a high-level, interpreted programming language with dynamic semantics. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. Python supports multiple programming paradigms, including procedural, object-oriented, and functional styles. It is widely used for web development, data science, and artificial intelligence applications [14] [15].

Lua: is a lightweight, multi-paradigm programming language designed for embedded systems and client-side applications. Lua is known for its simplicity, ease of embedding, and small footprint, making it ideal for scripting tasks and embedded systems

IV. Methodology

To evaluate the quality of code generated by LLMs in Python and Lua languages, the adopted methodology was based on the Test-Driven Development (TDD) approach. TDD is an agile software development process that emphasizes writing automated tests before writing the production code [8]. It was designed and developed a battery of unit-tests for each prompt; including automated performance tests and readability tests in the case of Python, in Lua this part was manual; time complexity tests.

For this study, the following LLM-based code generation tools were selected for comparison:

Table I: Summary of LLM-based code generation tools

Tool	Foundation Model	Domain
ChatGPT 3.5	GPT-3.5	General
GitHub Copilot	GPT4 [2]	Code
CodeLlama 34B Instruct	LLaMa 2	Code
Gemini	Gemini	General
Claude V2	Claude V2	General
Perplexity AI	unknown	General

Table II: Libraries used for each test metric in Python and Lua

Test Metric	Python	Lua
Accuracy	Pytest	Manual evaluation using If statements
Readability	Pylint	Luacheck extension
Complexity	Manual analysis[3]	Manual analysis
Performance	time.perf_counter()	time.clock()

A. Evaluation Metrics

The evaluation was performed considering the following core metrics:

- **Accuracy:** The concept of accuracy is closely linked to the notion of measurement error, which refers to the deviation from a true value [16]. In the context of this study, accuracy refers to the degree to which the generated code aligns with the intended functionality or approximation to the expected result of passing all unit tests [17] [18].
- **Readability:** Readability is the ease with which a developer can understand a program's source code. It is influenced by factors such as naming conventions, programming constructs, and formatting guidelines [19].
- **Performance:** Performance refers to the average time taken by the generated code to complete the desired task. In this study, the performance metric is calculated by measuring the average time taken by the generated code to execute its task over a number of iterations.
- **Complexity:** Complexity in this study refers to the algorithmic complexity of the generated code, i.e., the order of growth of its running time as the input size increases [20].

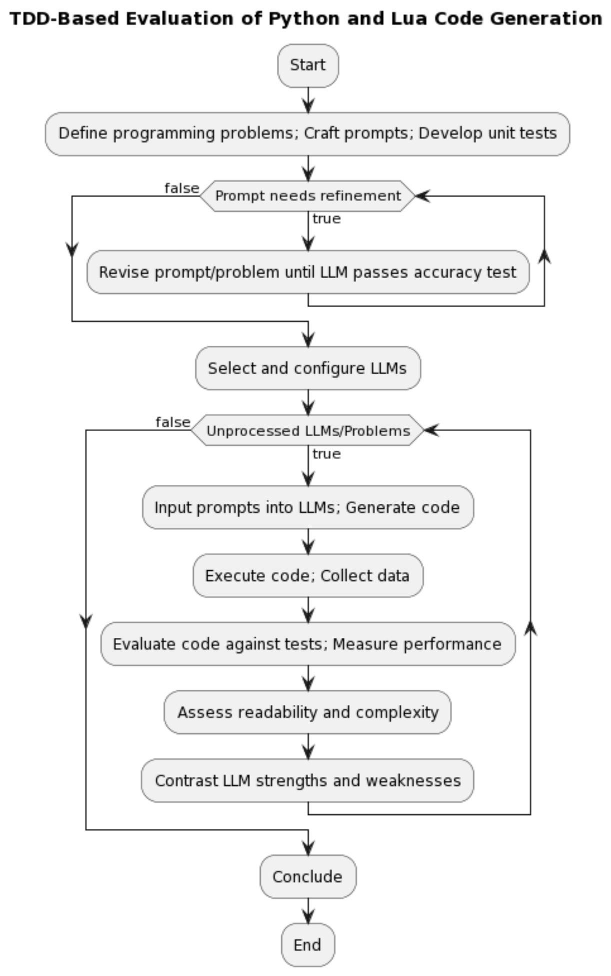
B. Libraries

The evaluation of the generated code was performed using a diverse set of tools for Python and Lua languages. This section presents the libraries used for each of the four test metrics, including accuracy, readability, performance, and complexity, along with a brief introduction to each tool's role. Table IV summarizes the libraries and their usage for the respective metrics.

For accuracy, *Pytest* was used in Python to run unit tests for different scenarios, while manual testing was employed

[2]As cited on the developer's blog: [4]

[3]Initially, the Big_O library was used for complexity analysis in Python.



TDD-Based Evaluation of Python and Lua Code Generation
Start
Define programming problems; Craft prompts; Develop unit tests
false
Prompt needs refinement
true
Revise prompt/problem until LLM passes accuracy test
Select and configure LLMs
false
Unprocessed LLMs/Problems
true
Input prompts into LLMs; Generate code
Execute code; Collect data
Evaluate code against tests; Measure performance
Assess readability and complexity
Contrast LLM strengths and weaknesses
Conclude
End

Figure 1: Evaluation pipeline for Python and Lua code generation

in Lua utilizing custom-built If statement conditions. In terms of readability, *pylint* was leveraged in Python to check the code against a set of rules and conventions, whereas *luacheck* Visual Studio Code extension served as the corresponding tool in Lua. For performance evaluation, the built-in Python function `time.perf_counter()` and its equivalent `os.clock()` in Lua were utilized to measure execution times. Finally, for assessing code complexity, Python's `Big_O` library was initially used but later replaced due to difficulties in handling certain edge cases; hence, manual analysis was conducted for both Python and Lua.

The performance tests were done by calculating the average runtime in milliseconds over 100k iterations, with a warm-up period prior to measuring the actual time.

1) Evaluation Pipeline: The evaluation pipeline followed the steps outlined below (Fig. 1):

1) **Define programming problems; Craft prompts; Develop unit tests:** Define a set of relevant programming problems in Python and Lua, write effective instruction prompts for each language model, and develop a suite of unit tests to cover various scenarios.
2) **Revise prompt/problem until LLM passes accuracy test:** Iteratively refine the prompts and problems until at least one or more of the language models is able to pass most or all the unit tests.
3) **Select and configure LLMs:** Select the language models to be evaluated, such as GitHub Copilot, Meta LLaMa 2, Gemini, Claude V2, Perplexity AI, and ChatGPT 3.5.
4) **Input prompts into LLMs; Generate code; Execute code; Collect data:** For each unprocessed LLM and programming problem, input the prompts, generate the code, execute the code, and collect the necessary data.
5) **Evaluate code against tests; Measure performance; Assess readability and complexity:** Evaluate the generated code against the unit tests to measure accuracy, conduct performance tests, and assess the readability and complexity of the code.
6) **Contrast LLM strengths and weaknesses:** Analyze the results to highlight the comparative strengths and weaknesses of the language models in generating source code for the defined programming tasks.
7) **Compute overall score:** Compute the overall score for each language model's output using the equation:

$$\text{score} = ((\text{A} \times 0.4) + (\text{R} \times 0.10) + (\text{P} \times 0.35) + (\text{C} \times 0.15)) \quad (1)$$

where *A* is Accuracy, *R* is Readability, *P* is Performance, and *C* is Complexity. Each factor is assigned a weight represented as a fraction of 1 (or 100%). For instance, *A* contributes 40% to the overall score, while *R* contributes 10%. A weight of 0 would imply the factor does not affect the score, and a weight of 1 (or 100%) would mean the factor completely determines the score.

The decision of the weights was given based on the importance of each metric in the context of code generation. Accuracy, for instance, is a fundamental aspect of code generation, as it ensures the code produced by the LLMs can perform the intended task. Readability, on the other hand, is important for maintainability and collaboration purposes, but not essential for the immediate functionality of the code. Performance is crucial for real-world applications, as it directly impacts the efficiency, while the scalability is dictated by the time complexity of the generated code. Therefore, the weights were assigned based on these considerations.

2) Calculating Scores: The scores for each LLM were calculated based on the evaluation metrics for accuracy, readability, performance, and complexity, as described in Section IV. The weights assigned to each metric are presented in Equation 1. The scores for each metric were normalized to a range of 0 to 1 before calculating the overall

Table III: Complexity score mapping

Complexity	Score
Constant	1.0
Logarithmic	0.85
Linear	0.70
Linearithmic	0.60
Quadratic	0.50
Cubic	0.30
Polynomial	0.20
Exponential	0.00

score.

The accuracy scores were obtained by calculating the percentage of unit tests passed for each code generation output i.e, given an x number of unit tests, if y number of tests pass, then the accuracy score would be y/x.

The readability scores were obtained by running the code through static code analysis tools such as Pylint for Python and Luacheck for Lua. These tools analyze the code and report violations of coding standards and best practices, assigning scores based on the severity and number of violations. The scores were then normalized to the range of 0 to 1, where a score of 1 represents perfect readability with no violations, and 0 represents poor readability with over 10 violations.

The performance scores were calculated by measuring the average execution time of the code over multiple iterations. The execution times were normalized to the range of 0 to 1, where the fastest code received a score of 1, and the slowest code received a score of 0.

The complexity scores were determined by manually analyzing the algorithmic complexity of the code, considering only the time complexity. The complexity scores were based on the following already normalized Table III:

Each code's complexity was manually assigned a score based on Table III.

V. Results

The results obtained through our evaluation process, with approximately 120 hours of local testing, were done on an Acer Nitro 5 laptop with a Ryzen 7 6800H processor, 16GB DDR5 4800MHz RAM, RTX 3070 TI graphics card, and Windows 11 Home OS.

Utilizing Python 3.10.8 and Lua 5.4.2, the evaluation results are presented in detail in Table IV and Table V. These tables display the final scores for each of the six LLMs tested in Lua and Python, respectively, after applying their individual results to the scoring Equation 1, where Accuracy, Readability, Performance, and Complexity were weighted as 40%, 10%, 35%, and 15%, respectively. The overall scores range from 0 to 1, with higher scores indicating better performance.

The evaluation results were extensively tested using various unit tests, which aimed to measure the accuracy of the LLMs' performance. A total of 16 unit tests were conducted, covering six different prompts. The results of these unit tests are presented below.

- **Prompt 1 (compare directories):** Two unit tests were conducted. The first test created two directories with all files being equal, and the second test created directories with a few files having different modification dates.
- **Prompt 2 (excel sort):** Three unit tests were performed. Each test checked the sorting of columns 1, 2, and 3, respectively, and verified the sorted data in the output file.
- **Prompt 3 (zipfs law):** Three unit tests were performed. The first test checked the frequency of the first six words, the second test checked the frequency of the 7th and 8th words, and the third test checked the frequency of the 9th and 10th words (see Section B-A for more details).
- **Prompt 4 (calculate stats):** Two unit tests were performed. The first test checked the calculated mean, and the second test checked the calculated standard deviation against expected results.
- **Prompt 5 (svg dot product):** Three unit tests were performed. The first test compared the calculated dot product with the expected result, the second test verified the return of -1 when there were fewer than two vectors in the metadata, and the third test verified the return of -1 when the vectors could not be parsed.
- **Prompt 6 (display text file):** Three unit tests were conducted. The first test assessed the successful execution of the implementation, returning True and a valid file path for an existing test.txt file. The second test examined the handling of non-existent or invalid file paths, verifying the return of False and None. The third test evaluated the correct sorting of name-value pairs in descending order according to their values.

In addition to accuracy, the performance of each LLM was evaluated by measuring the average execution time (in milliseconds) of the generated code over multiple iterations. The complexity of the generated code was manually analyzed based on the algorithmic complexity of each solution. Finally, the readability of the generated code was evaluated using static code analysis tools such as Pylint for Python and Luacheck for Lua. The scores for each metric were then normalized to a range of 0 to 1, with higher scores indicating better performance.

The results in Table IV demonstrate that the performance of the evaluated LLMs in generating Lua code was generally suboptimal. ChatGPT 3.5 emerged as the top-performing model, with only two scores below 0.5. In contrast, LLaMa, Perplexity AI, and Gemini (Bard) exhibited the lowest overall scores, with a few exceptions, such as LLaMa's performance on Prompt 5, and Perplexity AI's and Gemini's

[3]Note: P1-P6 represent Prompts 1-6. The highest possible score is 1.00.

Table IV: Ranking of LLM's with source code generated to Lua language

LLM	P1	P2	P3	P4	P5	P6	Avg
ChatGPT 3.5	0.755	0.078	0.101	0.561	0.635	0.929	0.509
Claude V2	0.067	0.066	0.667	0.538	0.416	0.011	0.294
CoPilot	0.067	0.011	0.462	0.944	0.755	0.579	0.469
Gemini (Bard)	0.0	0.0	0.011	0.944	0.022	0.0	0.162
LLaMa	0.089	0.044	0.146	0.089	0.722	0.0	0.181
Perplexity	0.0	0.256	0.438	0.944	0.602	0.0	0.373

Table V: Ranking of LLM's with source code generated to Python language

LLM	P1	P2	P3	P4	P5	P6	Avg
ChatGPT 3.5	0.531	0.501	0.856	0.908	0.584	0.575	0.659
Claude V2	0.654	0.455	0.798	0.86	0.6	0.586	0.658
CoPilot	0.622	0.508	0.251	0.859	0.756	0.908	0.65
Gemini (Bard)	0.655	0.506	0.748	0.305	0.588	0.55	0.558
LLaMa	0.376	0.025	0.751	0.919	0.198	0.328	0.432
Perplexity	0.000	0.523	0.890	0.06	0.795	0.877	0.524

performance on Prompt 4 [4].

The results showed in Table V reveal a more varied performance across the LLMs in generating Python code compared to their Lua counterparts. While no single model consistently outperformed the others across all prompts, ChatGPT 3.5 and GitHub Copilot exhibited the highest overall scores, with more instances of scores above 0.5. Notably, Perplexity AI demonstrated strong performance on Prompts 3, 5, and 6, indicating its capability in specific coding tasks.

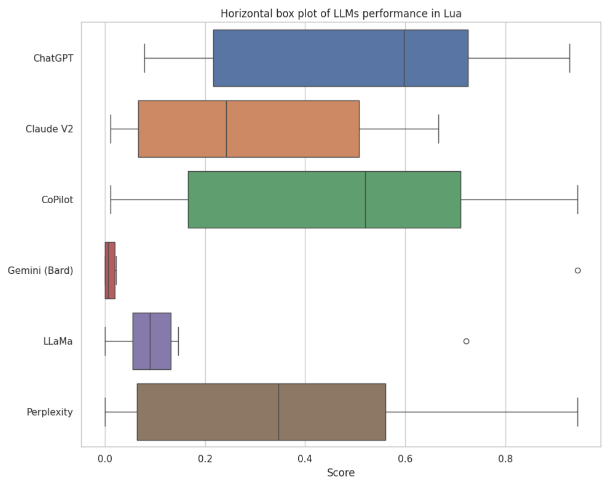
A. *Performance Overview*

The key findings derived from the analysis of the results are as follows.

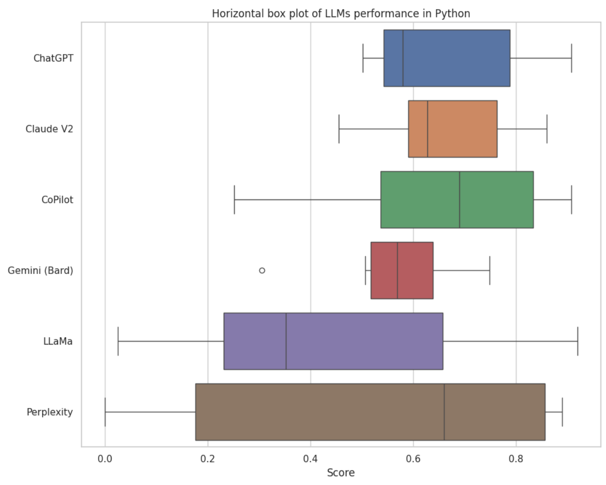
1) *Prompt-wise Analysis:*

- **Prompt 1: Compare Directories**
  For this prompt, **Claude V2** and **Gemini (Bard)** performed well in Python, while **ChatGPT 3.5** led the way in Lua. The complexity of this prompt, involving file system operations and comparing directory contents, posed challenges for some models.
- **Prompt 2: Spreadsheet Sort**
  This prompt proved to be challenging for most models, with **Perplexity** and **ChatGPT 3.5** achieving the highest scores in Python and Lua, respectively. The requirement to handle mixed data types and sorting logic may have contributed to the difficulty.
- **Prompt 3: Zipf's Law**
  Perplexity and **ChatGPT 3.5** excelled in this prompt for Python, while **Claude V2** led the pack for Lua. The models demonstrated proficiency in text processing and frequency analysis tasks.
- **Prompt 4: Calculate Statistics**
  Gemini (Bard), **CoPilot**, and **LLaMa** performed exceptionally well in this prompt for both languages, showcasing their strengths in mathematical computations and handling numerical data.
- **Prompt 5: SVG Dot Product**
  CoPilot and **LLaMa** achieved the highest scores in Lua, while **Perplexity** led the way in Python. The models exhibited varying degrees of success in parsing and manipulating SVG files.
- **Prompt 6: Display Text File**
  ChatGPT 3.5 and **CoPilot** excelled in this prompt for Python, demonstrating proficiency in file handling, data manipulation, and GUI development. However, most

[4]Note about scores below 0.1 in both Lua and Python: See V-A2 for more details



(a) Lua scores



(b) Python scores

Figure 2: scores per LLM on each prompt for Lua (left) and Python (right)

models struggled with this prompt in Lua, potentially due to language-specific libraries or the complexity of the requirements.

2) Outliers and Errors: During the evaluation process, several instances of outliers and errors were identified, as show by the scores in the Tables IV and V. These can be attributed to a variety of factors; here's a detailed analysis per LLM for each score below 0.1:

- **ChatGPT 3.5** The outlier occurred in prompt 2 (Excel Sort) for Lua, the generated code imported a placeholder excel_library, which made unit testing, performance profiling and code complexity analysis not viable for this scenario; leaving only the readability score with weight of just 10% to be computed.

-**Claude V2** In Lua, it had low scores for prompts 1, 2 and 6. For prompt 1 and 6, it had runtime errors due to calling a nil value; for prompt 1 it attempted to use ipairs(list_files(dir1)), where list_files was supposed to be a function that returns the list of files in dir1, yet since this function was not properly defined or implemented in the generated code, the error occurred, Claude V2 should have utilized one of the available libraries such as luafilesystem (lfs) for achieving the same task. In Prompt 6, it attempted to set up a GUI interface without actually importing any GUI related libraries, leading to a runtime error when attempting to call gg.newWindow(800, 600) expression, which should be used only after importing the respective GUI module. In both scenarios, all other metrics could not be computed, and only the readability score was determined; for prompt 2, Claude V2 got confused and imported xlsx instead of luaxlsx, which led to the execution failing as no xlsl.dll and xlsl.lua could be found inside LuaRocks packages, preventing it from finding the required module, the only score that could be calculated was based on readability.

- **GitHub Copilot** In Lua both prompt 1 (Compare Directories) and prompt 2 (Excel Sort) had runtime errors as they attempted to call a nil value[5], halting the execution of all further unit tests: for prompt 1 the generated code attempted to use a non-existent function table. Contains, and for prompt 2 it tried to access an undefined function load_excel_data for loading the Excel file at the start of the function. Both had their entire score based solely on their readability, since no other metrics could be calculated.

- **Gemini (Bard)** In Lua, the model had low scores for all prompts except prompt 4 (Calculate Stats). For prompt 2 (Excel Sort), it refused to write the code, claiming to be unable to directly create Lua code and only provided an outline of the steps needed to solve the problem; For Prompt 1, 3 and 6 it had runtime errors for calling a nil value; for prompt 1 it attempted to use a non-existent io function dir, for prompt 3 and 6 it tried to access a non-existent string function split; for prompt 5 the problem was the choice of library for the XML parsing, it has chosen the LuaXML library which is not compatible with our test environment (Lua 5.4.2) as it requires Lua 5.1 to 5.3 [21]. All these errors prevented the execution of the test suite, making it impossible to compute accuracy, performance, and complexity scores. Instead, these scores were determined solely based on readability.

- **LLaMa** It had four low scores in Lua: prompt 1 (Compare Directories), prompt 2 (Excel Sort), prompt 4 (Calculate Stats) and prompt 6 (Display Text File). For prompt 1, 4 and 6, the results were similar to those discussed above for ChatGPT 3.5 and GitHub Copilot; calling a nil value halted the execution of the test suite, making it impossible to compute accuracy, performance, and complexity scores. Instead, these scores were determined solely based on readability. For prompt 2, the model also produced a non-executable solution that failed to import the required library luaxlsx, this library has just a README in Chinese on GitHub, being maintained by xiyoo0812 and not supported by Luarocks for easy installation [22]. All four had their score solely based on their readability.

In Python, the prompt 2 (Excel Sort) had a Type Error due to an incorrect attempt to access a DataFrame column using the df.columns[column_index] expression inside the sort_values function, instead of using df[column_index], which would have worked correctly. This error prevented the execution of the test suite, leaving only the readability score to be computed.

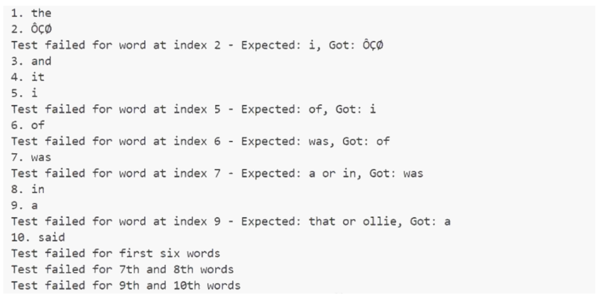
- **Perplexity AI** It had outliers in prompts 1 and 6 in Lua. For prompt 1 (Compare Directories), Perplexity AI encountered an infinite loop and an error due to the incorrect usage of io.popen with the external command ls, which doesn't exist on Windows, again, utilizing the lfs standard library of Lua would have prevented such issues as it works regardless of the operational system. For prompt 6, it had a variety of issues such as the usage of the json Lua library that only works up to Lua 5.3, which is not compatible with our test environment (Lua 5.4.2) [23], it made reference to a non-existent or incorrectly named fs library for handling files instead of the lfs library. All these errors prevented the execution of the test suite, making it impossible to compute accuracy, performance, and complexity scores, leaving only the readability score.

The results presented in Figure 3 show a stark contrast between the performance of ChatGPT and Claude V2 when executing Prompt 3 in Lua. The unit tests for this task consisted of 3 subtests:

- Test 1: First six-words frequency.
- Test 2: 7th and 8th words' frequencies.
- Test 3: Last two words' frequencies.

Test 1 makes sure that the top six most frequent words are

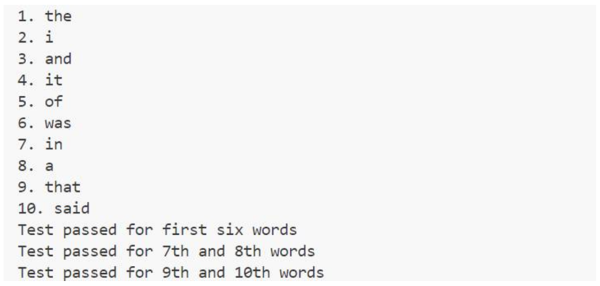
[5]In Lua, a nil value error occurs when a program attempts to use or access a variable or function that has not been assigned a value or has been explicitly set to nil. This error typically indicates that the program is trying to call a non-existent function or access a non-existent table key, leading to a runtime error and halting the execution of the program.



```
1. the
2. ÔÇØ
Test failed for word at index 2 - Expected: i, Got: ÔÇØ
3. and
4. it
5. i
Test failed for word at index 5 - Expected: of, Got: i
6. of
Test failed for word at index 6 - Expected: was, Got: of
7. was
Test failed for word at index 7 - Expected: a or in, Got: was
8. in
9. a
Test failed for word at index 9 - Expected: that or ollie, Got: a
10. said
Test failed for first six words
Test failed for 7th and 8th words
Test failed for 9th and 10th words
```

A

(a) ChatGPT results with 0% accuracy.



```
1. the
2. i
3. and
4. it
5. of
6. was
7. in
8. a
9. that
10. said
Test passed for first six words
Test passed for 7th and 8th words
Test passed for 9th and 10th words
```

B

(b) Claude V2 results with 100% accuracy.

Figure 3: Results of executing Prompt 3 in Lua (Zipf's Law).

identified correctly; Test 2 and 3 were made as there was more than one valid solution for the task, and they were designed to check the accuracy of the model in identifying the 7th and 8th most frequent words and the last two words, respectively.

In Figure 3 (A), ChatGPT implementation of Zipf's Law for Lua returned broken characters, such as *ÔÇØ* instead of i on the 2nd word, resulting in failure for the first test as one of the first six words was not identified correctly, for the 2nd and 3rd tests, the model failed to identify the 7th and 10th most frequent words correctly, it did not identify the 7th word as 'a', or 'in' but 'was' instead, and it did not identify the 10th word as 'that' or 'ollie' but 'a' instead. The model had 0% accuracy for this prompt, as it didn't return any valid results for the three established tests.

On the other hand, Figure 3 (B) shows the results from Claude V2, which achieved 100% accuracy for the same task as it correctly identified the top six most frequent words, the 7th and 8th most frequent words, and the last two words as required by the unit tests. Claude V2 demonstrated a much higher level of accuracy and proficiency in executing Prompt 3 in Lua, as it passed all the subtests and showed no signs of failure in identifying the correct words.

VI. Conclusion

The results of this study provide valuable insights into the strengths and weaknesses of various LLMs in generating source code for Python and Lua languages. It highlights the varying performance of these models across different programming tasks and language contexts.

While no single LLM consistently outperformed the others across all prompts, certain models exhibited notable strengths in specific areas. For example, ChatGPT 3.5 and CoPilot demonstrated proficiency in text processing, mathematical computations, and file handling tasks, while Claude V2 and Perplexity excelled in tasks involving frequency analysis and data manipulation.

As the field of generative AI continues to evolve, it is essential to maintain a critical and analytical approach to evaluating the performance and applicability of LLMs in software development. Ongoing research and collaboration between developers, researchers, and AI experts will be key to harnessing the full potential of these technologies while mitigating potential risks and ensuring responsible and ethical use.

VII. Supplementary information

All the material used to build this work is available in the GitHub repository indicated by the following link, including the texts of the prompts, generated source codes and test automation codes. This material is available for collaboration and improvements.

https://github.com/3750gustavo/TDD_Can_LLM_Code_unit_tests

References

[1] Wanlun Ma et al. *The "code" of Ethics:A Holistic Audit of AI Code Generators*. 2023. DOI: 10.48550/arxiv.2305.12747. arXiv: 2305.12747 [cs.CR].

[2] "Investigating and Designing for Trust in AI-powered Code Generation Tools". In: *arXiv.org* (2023). DOI: 10.48550/arXiv.2305.11248.

[3] "Generative AI carries serious online risks". In: (2023). DOI: 10.1108/oxan-db278161.

[4] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. "Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. <conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 172–184. DOI: 10.1145/3611643.3616271.

[5] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2023. DOI: https://doi.org/10.48550/arXiv.2308.12950. arXiv: 2308.12950 [cs.CL].

[6] Loredana Caruccio et al. “Claude 2.0 Large Language Model: tackling a real-world classification problem with a new Iterative Prompt Engineering approach”. In: *Intelligent Systems with Applications* (2024), p. 200336. DOI: https://doi.org/10.1016/j.iswa.2024.200336.

[7] Aram Bahrini et al. “ChatGPT: Applications, opportunities, and threats”. In: *2023 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE. 2023, pp. 274–279. DOI: https://doi.org/10.1109/SIEDS58326.2023.10137850.

[8] K. Beck. *Test-driven Development: By Example*. Sebastopol, CA: Addison-Wesley, 2003.

[9] Zhiqiang Yuan et al. “Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation”. In: (Aug. 2023). DOI: https://doi.org/10.48550/arXiv.2308.01240.

[10] Zhijie Liu et al. “No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT”. In: (Aug. 2023). DOI: https://doi.org/10.48550/arXiv.2308.04838.

[11] Li Zhong and Zilong Wang. “A Study on Robustness and Reliability of Large Language Model Code Generation”. In: (Aug. 2023). DOI: https://doi.org/10.48550/arXiv.2308.10335.

[12] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[13] Doron Haviv, Alexander Rivkind, and Omri Barak. *Understanding and controlling memory in recurrent neural networks*. PMLR, 2019.

[14] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[15] Gregory Piatetsky. “Python continues to eat away at R, RapidMiner gains, SQL is steady, Tensorflow advances pulling along Keras, Hadoop drops, Data Science platforms consolidate, and more”. In: *KDnuggets* (2018). URL: https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html.

[16] Stephen V Stehman, Giles M Foody, et al. *Accuracy assessment*. 2009.

[17] William M Stallings and Gerald M Gillmore. “A note on “accuracy” and “precision””. In: *Journal of Educational Measurement* 8.2 (1971), pp. 127–129.

[18] Mathias Hofer et al. “Definition of accuracy and precision—evaluating CAS-systems”. In: *International Congress Series* 1281 (2005). CARS 2005: Computer Assisted Radiology and Surgery, pp. 548–552. ISSN: 0531-5131. DOI: https://doi.org/10.1016/j.ics.2005.03.290. URL: https://www.sciencedirect.com/science/article/pii/S0531513105005492.

[19] Delano Oliveira et al. “Evaluating code readability and legibility: An examination of human-centric studies”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 348–359.

[20] Chat Room. “Algorithmic Complexity”. In: *algorithms* 16.05 (2022), p. 04.

[21] n1tehawk. *LuaXML*. https://github.com/n1tehawk/LuaXML. [Accessed 04-04-2024]. 2015.

[22] xiyoo0812. *luaxlsx*. https://github.com/xiyoo0812/luaxlsx. [Accessed 04-04-2024]. 2023.

[23] rxi. *json.luam - A lightweight JSON library for Lua*. https://github.com/rxi/json.lua. [Accessed 04-04-2024]. 2020.

[24] Seung Ki Baek, Sebastian Bernhardsson, and Petter Minnhagen. “Zipf’s law unzipped”. In: *New Journal of Physics* 13.4 (2011), p. 043004. DOI: 10.1088/1367-2630/13/4/043004.

sn-bibliography

Appendix A.
Prompt Engineering

To ensure that the language models produce relevant solutions, it was crucial to provide clear and well-structured instruction prompts. These prompts contained:

- A description of the task or problem, including any specifications such as input types, output requirements, or constraints.
- The target language (either Python or Lua).
- The expected behavior of the solution, including edge cases and possible scenarios.

The following prompts describe the problems submitted to each LLM for the Python programming language. The only differences between the Python and Lua versions of these prompts are the language name and the omission of any library-specific details that do not apply to both languages.

Prompt 1: “Write a Python function ‘compare_directories’ that compares two given directories (‘dir1’, ‘dir2’) and returns a detailed report of the comparison as a dictionary. The function should meet the following requirements:
Input Parameters: The function accepts two arguments:

- Directory 1 (dir1): A path to the first directory to be compared.
- Directory 2 (dir2): A path to the second directory to be compared.

Both directories contain similar sets of files, but potentially with differences in content and last modified dates.

Output Report Structure: The function generates a detailed report summarizing the outcome of the comparison. The report consists of a dictionary with the following key-value pairs:

- 1. Passed: This value is True if all tests pass, and False if any test fails.
- 2. Failed_Count: The total count of failed tests.
- 3. Failed_Tests: A list of detailed descriptions of failed tests[6].

PS: The Failure_Location is the directory with the oldest modification date for the files that failed the test, i.e., 'dir1' if the file with the same name in dir1 has an older modification date than the file in dir2."

Prompt 2: "Write a Python function called 'excel_sort' with two parameters: an Excel file path and a column index number. This function will use the pandas library to sort the data based on the specified column in descending order. The function must save to the disk temp file already with the sorted data and return it's path. PS: It should handle columns with mixed data types, handle sorting of strings, int, floats, dates, etc."

Prompt 3: "Write a Python function named 'zipfs_law' that accepts a string as an argument. This function should apply Zipf's Law [24] to find the top 10 most frequently used words in the string and return these words as a list in descending order. For that, you should remove all punctuation and convert all words to lowercase before counting them."

Prompt 4: "Write a Python function named 'calculate_stats' that accepts three lists of numbers as input. This function should calculate the mean and standard deviation for each list using the population standard deviation formula, and return these values as a dictionary with two keys: ''mean'' and ''stddev''. Each key should correspond to a list containing the mean and standard deviation values for the input lists, respectively. Be sure to include the necessary imports before the function definition."

Prompt 5: "Write a Python function named 'svg_dot_product' that receives a SVG file path as a parameter. This function should return the dot product of two vectors found within the SVG's <metadata>. The vectors are represented as text content of <vector> elements (e.g., <vector>1,2,3</vector>). If there are fewer than two vectors in the metadata or if they cannot be parsed, the function should return -1."

Prompt 6: "Develop a Python function named 'display_text_file' that accepts a single parameter representing the path to a .txt file. This function must perform the following tasks:

[6]Each dictionary in this list contains the following key-value pairs: Failed_Files: A list of file names that failed the test. Failure_Location: A string specifying where the failed test occurred, i.e., 'dir1', 'dir2'.

- Read the content of the provided .txt file.
- Utilize the delimiter ":" to parse the data within the file into name-value pairs with keys 'name' and 'value', respectively, as a list of dictionary variables.
- Sort the list of dictionaries in descending order based on the 'value' key.
- Check if a file named 'test.txt' already exists in the working directly, if not, create a temporary (Do not delete the file upon completion) .txt file named 'test.txt' with the sorted name-value pairs, each pair separated by a line break; if it does, overwrite the file with the sorted name-value pairs.
- Display the contents of the temporary .txt file in a window GUI.
- Upon completion, return a tuple containing a boolean value indicating whether the operation was successful and the absolute path to the temporary .txt file.
- If an error occurs during any of these steps, like the provided file path does not exist, the function should return false, NONE as the absolute path to the temporary .txt file and display an appropriate error message on the screen.

The function should be named 'display_text_file' and should be able to handle exceptions and errors gracefully."

APPENDIX B. STRENGTHS AND WEAKNESSES OF LLMS

- **ChatGPT 3.5** demonstrated consistent performance across most prompts, excelling in Prompt 3 (Zipf's Law) and Prompt 4 (Calculate Statistics) for both Python and Lua. However, it struggled with Prompt 2 (Excel Sort) in Lua.

- **Claude V2** demonstrated consistent performance across prompts, with notable strengths in Prompt 3 (Zipf's Law) for both languages and Prompt 1 (Compare Directories) for Python.

- **GitHub Copilot** delivered strong results in Prompt 4 (Calculate Statistics) for both languages and Prompt 6 (Display Text File) for Python, showcasing its capabilities in mathematical computations and file handling.

- **Gemini (Bard)** displayed varying results, with its highest scores in Prompt 4 (Calculate Statistics) for both languages and Prompt 1 (Compare Directories) for Python.

- **LLAMa** exhibited mixed performance, performing well in Prompt 3 (Zipf's Law) for Python and Prompt 5 (SVG Dot Product) for Lua but struggling with other prompts, particularly Prompt 6 (Display Text File) in Lua.

- **Perplexity** exhibited polarized performance, excelling in Prompt 3 (Zipf's Law) and Prompt 6 (Display Text File) for Python but failing to generate code for Prompt 1 (Compare Directories) in both languages.

A. Problems found along the evaluation process

While conducting the evaluation, several problems were encountered, including:

- **Bard refusal:** Gemini (formerly known as Bard) refused to write the excel_sort in Lua for prompt 2. It claimed to be unable to directly create Lua code and only provided an outline.
- **Lua libraries incompatibilities with anything newer than 5.2** : Many of the libraries utilized in the LLM's outputs, such as luaxlsx and LuaXML, had incompatibility issues with Lua versions above 5.2.
- **Non-existing Lua libraries**: The outputs of many LLMs included Lua code that referenced libraries or functions that did not exist in Lua, such as perplexity attempt to use fs instead of lfs (Lua File System) in prompt 6.
- **Placeholders instead of actual code**: Some outputs from LLMs, particularly ChatGPT in prompt 2 (excel_sort) and most LLMs in prompt 6 (display_text_file) in Lua, produced code that contained placeholders, like importing a generic nonexistent excel library and leaving a comment to be replaced by a real one in the case of ChatGPT. Most LLMs decided to leave not implemented the GUI elements of prompt 6 with the only exception of perplexity that tried, to implement using LÖVE library, but failed to even execute out of many errors with non-existing libraries.

These problems highlight the importance of using version-specific libraries, conducting thorough testing of the generated code, and refining the prompt engineering process to ensure that the LLMs are providing accurate, functional, and relevant code snippets for the specified tasks. Further research should focus on addressing these challenges and improving the overall quality and usability of LLM-generated code.

APPENDIX C. PROMPT CHANGELOGS

During the evaluation process, several changes were made to the prompts to clarify their expectations, improve their clarity, and ensure that they accurately reflected the unit tests and their requirements. Below, we outline the key changes made to each prompt:

- **Prompt 1:** The prompt for 'compare_directories' function was initially written as a high-level description of the desired functionality. Subsequent revisions provided a clearer explanation of the expected input, output, and behavior of the function, particularly in terms of how the function should handle files with identical names but different content or last modified dates.
- **Prompt 2:** Changes to the 'excel_sort' prompt primarily aimed to provide additional clarity on the expected output, making clear that the function should save the sorted data to a disk temp file and return the file path. The revisions also emphasized the need for the function to handle columns with mixed data types, such as strings, integers, floats, and dates. These revisions helped ensure that the generated code aligned with the specific requirements outlined in the unit tests.
- **Prompt 3:** No significant changes were made to the 'zipfs_law' prompt. A single revision related to the function flow of operation, specifically asking for the removal of punctuation and conversion of all words to lowercase before counting them, as the original prompt did not define if the input text should be cleaned or not. This clarification aimed to provide more precise guidelines for the LLMs on how to process the input string.
- **Prompt 4:** For the 'calculate_stats' prompt, the key revision was to explicitly state that the standard deviation should be calculated using the population standard deviation formula, instead of leaving it open to interpretation.
- **Prompt 5:** Only one change was made to the 'svg_dot_product' prompt: the inclusion of the case where there are fewer than two vectors in the metadata, making the final prompt include two scenarios where the function should return -1.
- **Prompt 6:** For the 'display_text_file' prompt, only one revision was made, with the sole purpose of expanding the descriptions of each requirement with examples for better clarification. For instance, it was made explicit that sorting should be done by the 'value' key, and that a new 'test.txt' temporary file should be created if not present.

These changes demonstrate the importance of iterative refinement during the evaluation process. By rephrasing and clarifying the prompts, we aimed to ensure that the generated code accurately reflected the expectations and requirements of the unit tests, ultimately improving the reliability and consistency of the results. Future studies can benefit from these lessons by incorporating prompt engineering into their methodologies from the outset, thus enhancing the overall quality and applicability of LLM-generated source code.

APPENDIX D. PARAMETERS USED FOR LIBRARIES ON TESTS

For *Pylint*, the version 3.1.0 and the default configuration were used, with no custom rule sets and no additional argument passed to it. For *Luacheck*, version 1.0.0 from the Visual Studio Code extension was utilized, employing the default settings without any custom configurations or additional arguments.

In the context of performance evaluation, the *time.perf_counter()* function in Python and *os.clock()* in Lua were employed to measure the average execution time over 100,000 iterations. This approach aimed to provide a reliable and consistent metric for comparing the computational efficiency of the generated code across different LLMs and programming languages. The warm-up period preceding the actual measurements ensured that any initial overhead, such as loading libraries or initializing environments, did not skew the results.

For the unit tests, *Pytest* version 7.4.3 was used to run the test suite in Python using as For the unit tests, *Pytest* version 7.4.3 was used to run the test suite in Python using the following command:

```
pytest -v -s absolute_test_file_path.py
```

For Lua, the unit tests were manually crafted using *if* statements to check the expected output against the actual output. The tests were designed to cover the same scenarios as their Python counterparts, ensuring that the evaluation of the generated code remained consistent across both languages.

In summary, the evaluation process was designed to be as comparable as possible, using default configurations for static analysis tools and standard practices for performance benchmarking and unit testing.