Evaluating Automated Test Case Generation in Python: A Comparative Analysis of LLM-Generated and LeetCode-Based Test Suites

Kushyanth Buddala
Department of Computer science
Colorado State University – Fort Collins, CO
Kushyanth.Buddala@colostate.edu
December 8th 2024

Abstract

Large Language Models (LLMs) have shown considerable potential in generating code, yet their effectiveness in producing comprehensive test cases for Python programming challenges remains underexplored. This study investigates the capabilities of ChatGPT-4 and Claude 3.5 in generating test cases compared to human-written baseline tests across varying difficulty levels. Using eighteen LeetCode problems evenly distributed across Easy, Medium, and Hard difficulties, test cases were generated through zero-shot and few-shot prompting strategies for both LLMs. Evaluation metrics included pass rates, execution time, cyclomatic complexity, and Codacy quality grades, with solutions validated against LeetCode-approved implementations. ChatGPT's few-shot approach generated the highest number of test cases (180+) with an average success rate of 73% across all levels, while Claude's zero-shot approach achieved the highest pass rate (90.48%) for Easy problems but performed poorly for Medium (51.61%) and Hard (60%) problems. Few-shot prompting enhanced ChatGPT's performance; however, it significantly reduced Claude's effectiveness to 41.67% for Medium problems. Codacy quality grades showed that both LLMs matched baseline quality for Easy problems but diverged significantly for harder tasks, with frequent F grades indicating quality concerns. These findings demonstrate that LLMs are effective for test case generation in Easy to Medium tasks, but human oversight remains critical for complex scenarios.

Keywords: Test Case Generation, Large Language Models, Automated Testing, ChatGPT, Claude, Python Testing, LeetCode.

1. Introduction

Software testing remains an essential yet time-intensive aspect of modern software development, ensuring the reliability and correctness of software systems. The advent of Large Language Models (LLMs) offers an innovative solution to streamline this process, particularly in automating test case generation for programming tasks written in Python. While LLMs have demonstrated remarkable potential in code generation, their effectiveness in producing comprehensive test suites remains insufficiently explored, especially when addressing problems of varying complexity levels.

This research focuses on evaluating the capabilities of two advanced LLMs ChatGPT-4 and Claude 3.5 for generating test cases for Python-based programming challenges. Among the available LLMs, including Google's Gemini, these two were specifically chosen for their proven track record in code generation and their stable, accessible APIs during the study period. The study aims to address a critical gap in research: understanding the effectiveness of LLMs in generating test cases across Easy, Medium, and Hard problems while comparing their output against human-written baseline tests. This comparison is essential to determine whether LLMs can reliably augment or, in specific scenarios, replace manual test case creation.

The significance of this research lies in its practical implications. With Python's widespread adoption in academia and industry, the demand for efficient test generation has become increasingly critical. Automated test case generation using LLMs could significantly reduce the time and effort required for testing, enabling developers to focus on more complex tasks. This study provides empirical evidence regarding the strengths and limitations of LLMs in automated testing, offering valuable insights for developers and researchers alike.

The research methodology involves systematically selecting eighteen Python-based LeetCode problems, evenly distributed across Easy, Medium, and Hard difficulties, with a focus on array and string manipulation tasks. Test cases were generated using two distinct prompting strategies—zero-shot and few-shot—for both ChatGPT-4 and Claude 3.5. Each test case was validated by running it against LeetCode-approved solutions to determine whether the test passed or failed, assessing the effectiveness of the generated test cases. To further evaluate the quality of the test cases, the study incorporated two key tools: Radon was used to measure cyclomatic complexity, providing insights into the complexity of the generated test cases, and Codacy was employed to assess code quality, assigning grades ranging from A to F. This comprehensive evaluation framework ensures a robust analysis of the generated test cases across multiple dimensions, including pass rates, complexity, and quality.

By examining these metrics, the study addresses critical questions: How effective are LLMs in generating diverse and reliable test cases? How do zero-shot and few-shot prompting strategies influence the quality of the tests generated? And how do LLMs perform in comparison to human-written baseline tests across varying problem complexities? The findings of this research aim to guide developers and organizations in integrating LLMs into their software testing workflows. While LLMs like ChatGPT-4 and Claude 3.5 show significant promise, the study highlights the importance of human oversight in complex scenarios and provides a foundation for future advancements in LLM capabilities for automated testing.

2. Approach

This study adapts existing approaches for automated test case generation to the Python programming domain, focusing on the capabilities of ChatGPT-4 and Claude 3.5. The approach leverages automated scripts to streamline problem selection, dataset creation, test generation, and evaluation. The scripts ensured consistency, reproducibility, and accuracy in the methodology.

2.1 Goal of the Evaluation

This study aims to evaluate the effectiveness of LLMs in generating test cases for Python programming challenges. The goals and corresponding metrics/tools used are as follows:

1. **Assess the reliability of LLM-generated test cases across different difficulty levels (Easy, Medium, Hard):**
    - **Metric:**
        - Pass/Fail Rates (A automated script made to run all the test files through the validated solutions and shows a overview of passed or failed test cases)
    - **Tool:**
        - Custom Python-based automation framework for executing test cases against LeetCode-approved solutions.
2. **Analyze how zero-shot and few-shot prompting strategies influence the quality and effectiveness of test cases:**
    - **Metric:**
        - Pass/Fail Cyclomatic Complexity (to measure test sophistication and maintainability) **and**
        - Codacy Quality Grades (to evaluate code quality based on adherence to standards and potential bugs)
    - **Tool:**
        - Radon for measuring cyclomatic complexity.
        - Codacy for automated quality grading on an A–F scale.

3. **Compare LLM-generated test cases with human-written baseline tests to identify strengths and weaknesses:**

    - **Metric:**
        - Pass/Fail Rates (to compare effectiveness) **and**
        - Cyclomatic Complexity, Maintainability and Codacy Grades (to compare test structure and quality).
    - **Metrics:**
    - **Tools:**
        - Radon and Codacy, as above, for detailed quality analysis.

These goals establish a comprehensive framework for evaluating the performance of ChatGPT-4 and Claude 3.5, enabling systematic comparisons between the LLMs, prompting strategies, and baseline human-written test cases.

2.2 Problem Selection and Dataset Creation

The foundation of this study rests on a carefully curated dataset of Python programming problems. Automated scripts were developed to implement the selection criteria, extract data, and structure the dataset for consistent analysis. These scripts ensured that each problem file included descriptions, metadata, solution templates (without the solution), test files, and complete solutions.

Selection Criteria and Dataset overview:

Focused on array and string operations due to their consistent testing patterns, clear complexity measures, and abundant problems on LeetCode. Eighteen problems were selected (six per difficulty level), evenly divided between array and string manipulation tasks.

Acceptance rate thresholds:

- Easy: >70%, Medium: >50% and Hard: >30%

A total of 90 test suites were generated:

- 18 human-written test suites.
- 72 generated test suites (18 problems × 2 LLMs × 2 prompting strategies).

Scripts stored the overall collected data in JSON format, including problem ID, difficulty level, problem category, and other metadata. The dataset provided standardized problems with verified solutions and human-written test cases, making it ideal for structured evaluation and comparison.

2.3. Test Generation Approaches

The study employed two distinct prompting strategies for generating test cases through API calls to ChatGPT-4 and Claude 3.5. A temperature of 0.2 was used to balance creativity and accuracy, with time delays between API calls to ensure stable results.

- **Zero-Shot Prompting:**
  This approach involved providing LLMs with only the problem description, a solution template (without the solution), and basic requirements for pytest-compatible test case generation. These requirements outlined essential structural elements, such as type hints, assertion patterns, and class organization, ensuring that the generated tests followed a consistent format. Unlike few-shot prompting, zero-shot prompting tested the LLMs' ability to independently infer the necessary test

cases without guidance or examples. The intent was to evaluate the baseline capability of the LLMs to understand problem descriptions and generate meaningful, original test cases using only the provided requirements. This approach avoided any form of direct instruction or template-based influence, ensuring that the results purely reflected the LLMs' inherent ability to reason and generalize from the problem description alone.

- **Few-Shot Prompting:**
  This approach expanded on the zero-shot strategy by incorporating carefully selected example test cases from problems of similar difficulty. For instance, when generating test cases for "Split a String in Balanced Strings" (Easy), templates from "Jewels and Stones" (another Easy string manipulation problem) were provided. These templates served as examples of test structure and organization, rather than specific solutions, and were chosen to demonstrate proper coverage of edge cases and diverse scenarios. While the requirements in zero-shot prompting provide basic formatting guidelines, the few-shot approach offers additional context on how these requirements translate into well-structured test cases. The intent is not to bias the LLM with problem-specific hints but to enhance its understanding of test design patterns, enabling it to produce higher-quality, more comprehensive test cases.

2.4. Evaluation Framework

The evaluation framework was designed to assess the generated test cases.

- **Primary Effectiveness**:
  Pass/fail rates were used as the primary metric to evaluate the effectiveness of the generated test cases. Each test case was executed against LeetCode-approved solutions to verify correctness. A custom automation framework standardized the execution environment and resolved common import issues to ensure consistency.
- **Code Quality**:
  Two tools were employed for code quality assessment:
  1. **Radon**: Measured cyclomatic complexity to evaluate test case sophistication and maintainability.
  2. **Codacy**: Assigned automated quality grades (A–F), considering factors such as code style, potential bugs, and general coding practices.
     This dual assessment approach provided a comprehensive view of the strengths and weaknesses of the generated test cases.

This dual assessment approach provided a comprehensive view of the strengths and weaknesses of the generated test cases. The visualizations for pass rates, cyclomatic complexity, Codacy grades, and execution times, generated from CSV data, provided a comprehensive assessment of the generated test cases. Pass rates and execution times compared the effectiveness and efficiency of LLM-generated test cases with human-written baselines, while cyclomatic complexity and Codacy grades assessed test quality and maintainability. These visualizations enabled a direct comparison of independent variables (LLM models, prompting strategies, difficulty levels) with dependent variables (pass/fail rates, code quality, efficiency), highlighting key performance trends across test generation approaches.

2.5 Variables

The study considered three categories of variables to ensure a comprehensive evaluation: independent, dependent, and state variables. These variables provided a structured framework for analyzing the capabilities of ChatGPT-4 and Claude 3.5 in generating Python test cases.

Independent Variables:

- **Test Generation Source:** ChatGPT-4, Claude 3.5, and human-written baseline (LeetCode).
- Problem Difficulty Level: Easy, Medium, and Hard.

Dependent Variables:

- **Effectiveness:** Measured using pass/fail rates of test cases.
- **Code Quality:** Evaluated through cyclomatic complexity (Radon), maintainability index, and Codacy grades (A–F).
- **Efficiency:** Execution time recorded during test runtime.

State Variables:

- **API Configuration:** Temperature set at 0.2 with time delays to ensure consistent responses.
- **Problem Selection Criteria:** Array and string manipulation problems with acceptance rates: Easy >70%, Medium >50%, Hard >30%.
- **Prompt Structure:**
    - Zero-shot: Provided only problem descriptions and basic requirements.
    - Few-shot: Included templates for structural guidance from similar problems.

3. Results

3.1 Effectiveness: Pass/Fail Rates

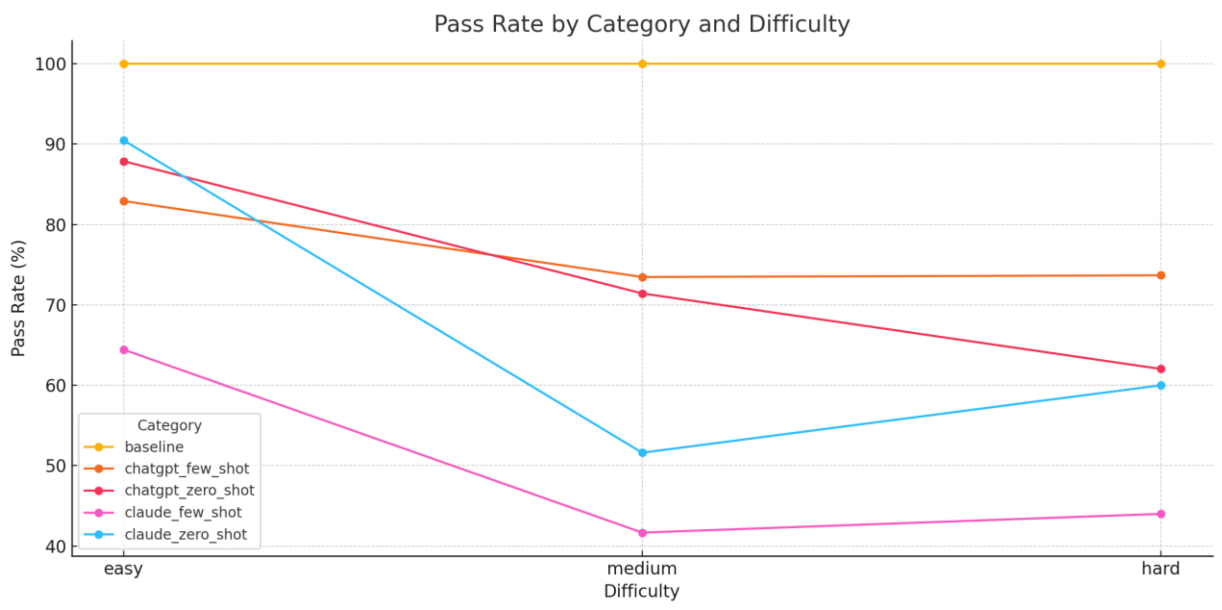
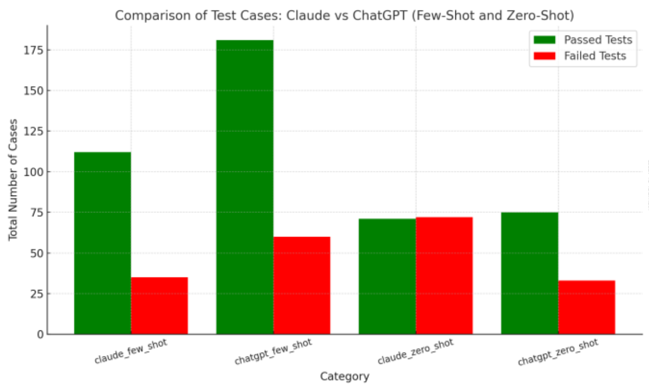


Fig 1 Pass Rate by Category and Difficulty



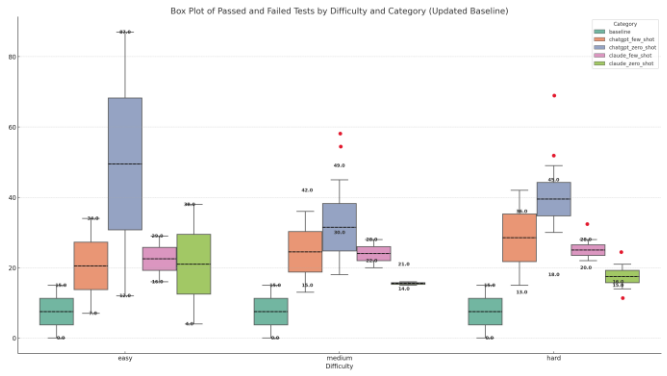


Fig 2 and Fig 3 Plots of Passed and Failed Tests

3.2 Code Quality: Cyclomatic Complexity and Codacy Grades

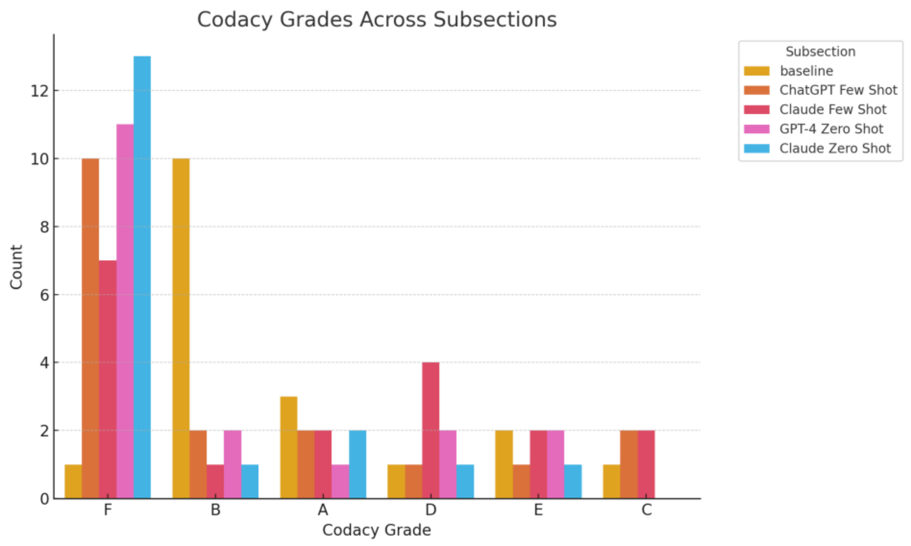


Fig 4 Distribution of Codacy grades (A–F)

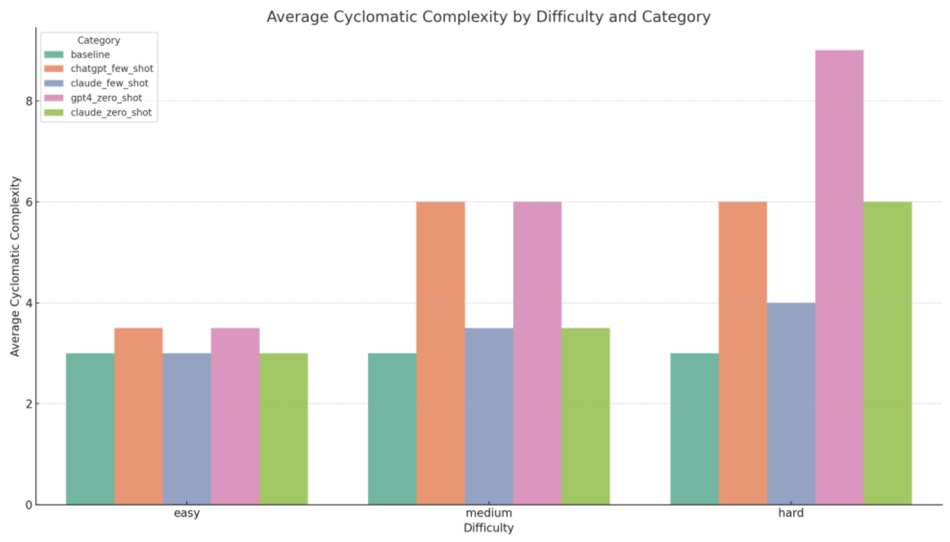


Fig 5 Average cyclomatic complexity of test cases

3.3 Efficiency: Execution Time

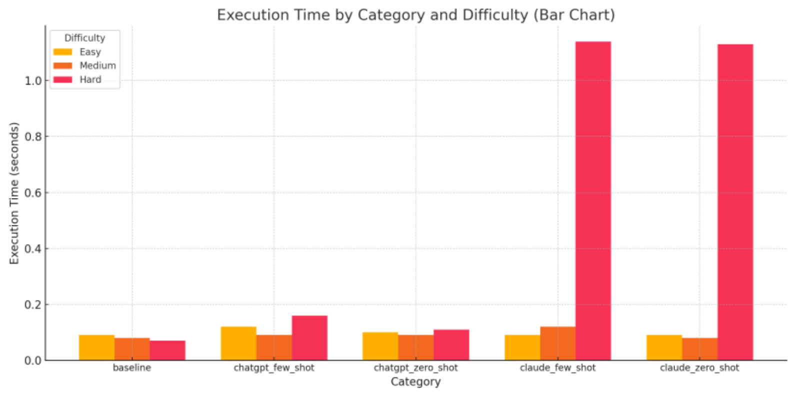


Fig 6 Execution Time by Category

3.4 Maintainability Index

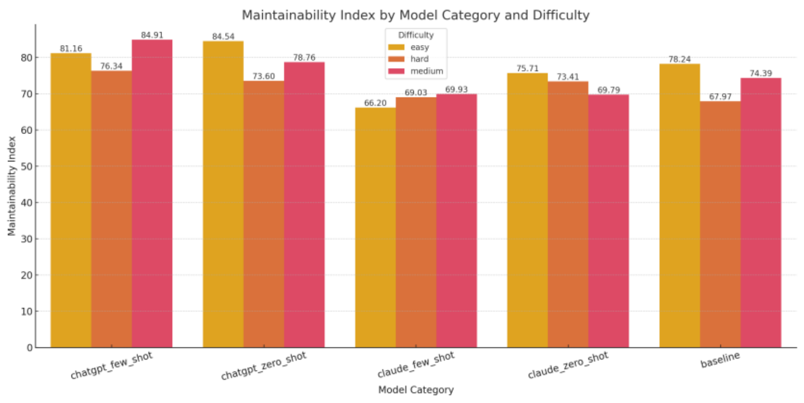


Fig 7 Maintainability Index

4. Discussions

The analysis of the results provides insights into the effectiveness of ChatGPT-4 and Claude 3.5 in generating Python test cases. I compared their performance across several metrics, including pass rates, cyclomatic complexity, execution time, and maintainability index. The results reveal both strengths and weaknesses in the LLM-generated test cases, highlighting trends that align with previous studies.

Pass Rates from Fig 1 indicate that ChatGPT-4 and Claude 3.5 performed well on Easy problems, with ChatGPT-4 maintaining an 87.88% pass rate and Claude 3.5 achieving 90.48%. However, as problem difficulty increased, both LLMs showed a marked decline in performance. For Medium problems, ChatGPT-4's pass rate dropped to 71.43%, and Claude 3.5's pass rate fell significantly to 51.61%, indicating a clear difficulty in generating reliable test cases for more complex problems. For Hard problems, Claude 3.5 struggled the most, with only 60% pass rate, while ChatGPT-4 was still able to achieve 62.03%. These trends align with Schäfer et al. [1], who observed that LLMs faced difficulties generating accurate tests as problem complexity increased, supporting the notion that LLMs excel at simpler tasks but struggle with more intricate scenarios.

The distribution of passed and failed tests (Fig 2 and Fig 3) further corroborates these findings. Across all categories, a total of 254 test cases were generated and evaluated, with ChatGPT-4 successfully passing 173 tests (68.1%) and Claude 3.5 passing 162 tests (63.8%), compared to the baseline's 100% pass rate of 48 test cases. Both models showed a higher number of failures for Medium and Hard problems, with Claude

3.5 demonstrating more significant failure rates, particularly for Hard problems. These failed tests were often characterized by outliers, indicating potential errors or cases where the test generation process encountered difficulties due to edge cases or the complexity of the problem. This mirrors the results from Guilherme and Vincenzi [3], where ChatGPT was also observed to perform well on simpler problems but struggled with more complex ones.

In terms of code quality, as shown in Fig 4, the Codacy grades revealed that ChatGPT-4 performed better than Claude 3.5, especially for Easy and Medium problems. ChatGPT-4 produced more A and B grades, indicating higher code quality, while Claude 3.5 had a larger share of C, D, and F grades, particularly for Hard problems. These findings are consistent with Chen et al. [2], who found that improved prompt techniques in ChatGPT led to better code quality. However, the Codacy grades for Claude 3.5 in more difficult problems highlight potential weaknesses in its test generation, with most of its tests failing to meet basic coding standards.

The cyclomatic complexity scores, displayed in Fig 5, reflect the structural complexity of the generated test cases. The Baseline tests had the lowest complexity, with an average score of 4.2 across all difficulty levels. ChatGPT-4 showed moderate complexity increases, with an average complexity of 5.7 for Easy problems, rising to 7.9 for Hard problems. This suggests that ChatGPT-4 generated more comprehensive tests, but at the cost of increased complexity. Claude 3.5 had the highest average complexity, with scores reaching 9.1 for Hard problems, indicating that it generated more intricate tests, but these were harder to maintain. This pattern is consistent with Schäfer et al. [1], where LLMs generated increasingly complex tests as difficulty increased, though this often led to tests that were harder to manage.

Regarding execution time (Fig 6), Baseline tests showed the shortest execution times, averaging between 0.07 to 0.09 seconds, indicating the efficiency of human-written tests. ChatGPT-4's execution times were slightly longer, averaging 0.12 to 0.16 seconds, likely due to the increased complexity of its generated test cases. Claude 3.5 exhibited the longest execution times, especially for Hard problems, where execution times exceeded 1 second. This suggests that Claude 3.5 generated less optimized test cases, resulting in slower execution. Similar inefficiencies were noted by Guilherme and Vincenzi [3], where LLM-generated tests had longer execution times, likely due to suboptimal code generation.

The Maintainability Index (Fig 7) provides insight into the long-term usability of the generated test cases, it shows that Baseline tests had the highest maintainability, with an average index of 85, indicating that human-written tests were simple and easy to maintain. ChatGPT-4 showed an average maintainability index of 78, which is still strong but slightly lower than the baseline, reflecting the complexity introduced in the test cases. Claude 3.5 had the lowest average maintainability index of 67, particularly for Hard problems, where the complexity of its generated tests made them harder to maintain. These results are consistent with Naimi et al. [4], who observed that more complex LLM-generated tests led to reduced maintainability.

The findings from this study reveal that ChatGPT-4 generally outperforms Claude 3.5 in generating effective and maintainable test cases, particularly for easier problems. However, both LLMs showed difficulty with more complex test cases, leading to lower pass rates, higher cyclomatic complexity, longer execution times, and reduced maintainability. These results align with previous studies, particularly Schäfer et al. [1], Chen et al. [2], and Guilherme and Vincenzi [3], which all noted that LLMs excel in easier test cases but face significant challenges with more complex ones. Human-written tests consistently outperformed LLM-generated ones across all metrics, underlining the importance of human oversight in generating high-quality, maintainable test cases.

5. Threats to Validity

Several threats to the validity of the study must be considered to assess the reliability of the results.

5.1 External Validity:

The study focused on 18 LeetCode problems, all based on array and string manipulations, which may not fully represent the diversity of programming challenges. While these problems are common in coding assessments, the results may not generalize well to more complex algorithmic tasks or other programming languages. Additionally, the performance of ChatGPT-4 and Claude 3.5 may differ on more specialized tasks, as the models may be better suited for certain problem types. Despite this, the study remains valuable for understanding LLM performance on typical LeetCode-style problems in Python.

5.2 Internal Validity:

The observed results could be influenced by factors unrelated to the prompting strategies. The temperature setting of 0.2, aimed at balancing creativity and accuracy, may have impacted the models' ability to generate diverse test cases. A higher temperature might have led to more varied outputs, possibly improving test quality. Additionally, the cross-problem templates used in the few-shot prompting might have limited the LLMs' ability to adapt to unique problem structures. Using problem-specific templates could yield different results, highlighting that the chosen prompting structure might have constrained the models' potential. These factors should be considered when interpreting the results, as they may have influenced the generated test cases.

5.3 Construct Validity:

The study measured pass/fail rates, cyclomatic complexity, execution time, and maintainability index to assess test case effectiveness and quality. While these metrics are standard for evaluating test cases, they may not capture all aspects of test case quality, such as edge case handling or test case adaptability. For instance, a high pass rate may indicate correctness but not necessarily comprehensive test coverage. Moreover, complexity and maintainability metrics may not fully reflect the real-world challenges of working with LLM-generated tests in production environments. A more detailed coverage analysis like line or branch coverage would likely provide a better understanding of the test case quality but was not included in this study due to time constraints.

In summary, while there are limitations due to the problem selection, temperature settings, and template structure, the study's results provide useful insights into the capabilities of ChatGPT-4 and Claude 3.5 in generating Python test cases. The findings highlight both strengths and weaknesses, suggesting areas for improvement in test case generation strategies.

6. Related Work

Schäfer et al. (2024)[1] evaluated LLM-based test generation using their tool, TESTPILOT, with OpenAI's GPT-3.5 to generate tests for JavaScript functions across 25 npm packages. Their structured approach incorporated function signatures and usage examples, achieving a median statement coverage of 70.2% and branch coverage of 52.8%. Complexity was analyzed using Radon. While this project does not replicate their coverage evaluation, it adapts their framework to assess test case effectiveness and complexity in a Python context. Specifically, Radon was used to analyze cyclomatic complexity, and pass/fail rates were employed to evaluate the functionality of LLM-generated test cases by ChatGPT-4 and Claude 3.5.

Chen et al. (2024)[2] introduced ChatUniTest, a framework for Java unit test generation that optimizes prompt quality through adaptive prompt engineering techniques, including zero-shot and few-shot prompting. Their study employed OpenAI's Codex for test generation and demonstrated that adaptive prompting increased test quality by 15% compared to standard methods. While this study does not adopt coverage analysis using JUnit or JaCoCo, it adapts Chen et al.’s prompt engineering methodology for Python test case generation. Codacy was used to evaluate the quality of LLM-generated test cases on an A-F scale, with comparisons to human-written LeetCode tests. In addition, Guilherme and Vincenzi [3] conducted an initial investigation into ChatGPT’s unit test generation capabilities. Their work explored various prompting techniques for Java functions and reported moderate success in generating syntactically correct tests. However, their evaluation did not extend to assessing the functional effectiveness or quality of the generated tests. This study is relevant as it highlights the role of prompt design, a central aspect of this research. Naimi et al. (2024)[4] proposed a novel method for automatic test case generation using LLMs and prompt engineering applied to use case diagrams. Their approach demonstrated how structured prompts could guide LLMs in generating tests for specific scenarios. While their study did not focus on programming tasks like Python functions, the insights into structured prompt design provide parallels with the few-shot prompting methodology employed in this project.

Through these adaptations and reviews, this project draws on complexity assessment methods from Schäfer et al.[1] and adaptive prompt engineering techniques from Chen et al.[2] to evaluate Python test cases generated by ChatGPT-4 and Claude 3.5. Additionally, while Guilherme and Vincenzi [3] and Naimi et al. [4] provide complementary insights, they are not directly adapted here. Instead, their work highlights the evolving landscape of LLM-based test generation and underscores the need for diverse evaluation frameworks, which this study addresses by emphasizing pass/fail rates, complexity scores, and Codacy grades.

7. Conclusion

The evaluation of ChatGPT-4 and Claude 3.5 for generating Python test cases revealed some key lessons about the capabilities and limitations of LLMs in test case generation. First, ChatGPT-4 demonstrated consistent reliability across all difficulty levels, especially excelling with Easy and Medium problems. Its performance was more stable, even for Hard problems, when compared to Claude 3.5, which experienced substantial performance degradation on more difficult tasks. This suggests that ChatGPT-4 may be better suited for real-world applications requiring a reliable test-case generation system for a wide range of problems.

A crucial lesson from the study is the importance of problem complexity. Both LLMs showed significant challenges in generating effective tests for more complex problems, with Claude 3.5 struggling particularly with Hard problems. This highlights the fact that, while LLMs can generate functional test cases for simpler problems, their ability to adapt to increasing problem complexity remains limited. Therefore, human oversight remains important, especially in scenarios involving intricate edge cases or advanced problem-solving requirements.

Additionally, the study highlighted the value of prompt engineering. The few-shot approach, where example templates were provided, helped ChatGPT-4 improve its test case generation. However, the benefit of few-shot prompting was less clear for Claude 3.5, as the model did not consistently improve with the inclusion of example templates. This suggests that the effectiveness of few-shot prompting may depend on the model's inherent capabilities and could benefit from more targeted fine-tuning. From a quality perspective, the results indicated that while LLMs performed reasonably well in generating tests for Easy problems, the complexity and maintainability of test cases increased as problem difficulty grew, often leading to lower quality scores, particularly with Claude 3.5. These findings emphasize that complexity needs to be carefully balanced with maintainability in automated test generation, and there may be diminishing returns in terms of quality as LLMs aim to cover more complex scenarios.

In conclusion, ChatGPT-4 has proven itself to be a more reliable option for generating test cases for Easy and Medium problems, while Claude 3.5 showed potential but needs further refinement, particularly for more complex tasks. Future work should explore strategies to improve the performance of LLMs in handling Hard problems and enhance their ability to generate maintainable, high-quality test cases.

References

[1] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation,” *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85-105, Jan. 2024, doi: 10.1109/TSE.2023.3334955.

[2] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin. “ChatUniTest: A Framework for LLM-Based Test Generation,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024)*, ACM, 2024, pp. 572-576.

[3] V. H. Guilherme and A. M. R. Vincenzi, "An initial investigation of ChatGPT unit test generation capability," *Federal University of São Carlos, Brazil*, 2024.

[4] L. Naimi, E. Bouziane, M. Manaouch, and A. Jakimi, "A new approach for automatic test case generation from use case diagram using LLMs and prompt engineering," *GL-ISI Team, Faculty of Sciences and Techniques of Errachidia, UMIMeknes, Morocco*, 2024.