A Case Study on Test Case Construction with Large Language Models: Unveiling Practical Insights and Challenges

Roberto F. Lima Jr[1], Luiz Fernando P. B. Presta[2], Lucca S. Borborema[2], Vanderson N. Silva[1], Marcio L. M. Dahia[1], Anderson Santos[1]

[1]CESAR
Recife, PE, Brazil

[2]CESAR School
Recife, PE, Brazil.

{rflj,vns, mlmd, acss}@cesar.org.br, {lfpbp, lsb}@cesar.school

Abstract. *This study delves into the integration of Large Language Models (LLMs) in test case construction within software engineering, exploring their potential to enhance efficiency and effectiveness in test generation. Leveraging LLMs, known for their sophisticated natural language processing abilities, this research conducts a detailed case study on a representative software application to evaluate the practicality of LLMs in creating detailed and accurate test scenarios. The investigation focuses on the challenges and advantages of LLMs in test case development, assessing their impact on test comprehensiveness, accuracy, and the formulation process. By providing a nuanced understanding of LLMs' role in software testing, this paper aims to inform practitioners and researchers about their potential and limitations, offering insights into their application in real-world testing environments and their contribution to advancing software testing methodologies.*

1. Introduction

Software testing constitutes a critical phase in the software development life cycle, ensuring the delivery of reliable and high-quality software products. In recent years, the advent of advanced technologies, particularly Large Language Models (LLMs), has introduced novel possibilities for automating and enhancing various facets of software engineering. LLMs, exemplified by models such as OpenAI's GPT (Generative Pre-trained Transformer)[OpenAI 2023], exhibit remarkable natural language processing capabilities and have demonstrated applicability in diverse domains. This paper addresses the intersection of LLMs and software testing, focusing specifically on their role in the construction of test cases.

The use of LLMs in software testing holds the promise of increasing the efficiency and effectiveness of test case generation. The ability of these models to understand and generate human-like text prompts paves the way for more natural and expressive articulation of test scenarios. However, the practical implications of integrating LLMs into the nuanced process of test case construction remain under explored. This study seeks to bridge this gap by presenting a detailed case study that investigates the application of LLMs in constructing test cases for a real-world software application.

The case study methodology adopted in this research involves the selection of a representative software application, the formulation of test case construction methodologies leveraging LLMs, and a thorough evaluation of the outcomes. By examining the interplay between LLMs and the complexities inherent in software testing, this study aims to provide insights into the practical advantages, challenges, and considerations associated with their implementation. Through this exploration, the intent is to contribute substantively to the ongoing dialogue surrounding the integration of cutting-edge technologies, such as LLMs, in the pursuit of optimizing software testing practices.

2. Related Work

The intersection of Large Language Models (LLMs) and software engineering has witnessed growing interest in recent literature, reflecting a broader acknowledgment of the transformative potential of these models in many domains. Within the realm of software testing, where the efficacy of test case construction significantly influences software quality, the exploration of LLM applications remains a relatively nascent but promising area.

The preeminent contribution of LLMs to software engineering lies in their ability to comprehend and generate human-like text, a feature that has inspired its application in diverse natural language processing tasks. Prior studies, such as [Brown et al. 2020] showcase the prowess of LLMs, exemplified by models like GPT-3, in understanding context, generating coherent responses, and adapting to diverse language inputs. This linguistic versatility positions LLMs as potentially powerful tools for expressing and formulating test cases in a more natural, context-aware manner.

Taking advantage of LLMs, many Generative AI models have been developed to implement diverse use cases, such as generating documentation for code [Feng et al. 2020], auto-completing code [Kim et al. 2021] and generating unit tests [Tufano et al. 2021].

In the specific context of software testing, recent research by [Wang et al. 2024] summarized the use of LLMs in a diverse set of applications, integrating them with traditional methods such as differential and mutation testing. Despite showing promise, LLMs are recognized as complementary rather than exclusive solutions for software testing. The research also combined LLMs with metamorphic testing, generating test cases based on expected input-output relationships, and explored their role in model-based testing, enhancing model accuracy through natural language descriptions and adapting to system changes.

However, existing literature lacks comprehensive insights into the practical implications, challenges, and considerations associated with integrating LLMs into the intricate task of test case construction for real-world software applications.

3. Approach

This section outlines our approach, consisting of the semi-automated test cases constructions and the evaluation framework for it. First it was explored the integration of LLMs with software specifications for generating test cases and later elucidates our method for qualitatively assess the effectiveness of the generated test cases.

3.1. Test cases construction

To generate test cases from the model, the strategy of prompt engineering was employed. The prompt, which serves as the input for the LLM, comprises a set of instructions and necessary data for task completion. The quality of the model's response improves with clearer and more contextual descriptions provided in the prompt. Another effective strategy for achieving positive outcomes involves allowing the model to engage in reasoning by decomposing the task into intermediate steps [Wei et al. 2022].

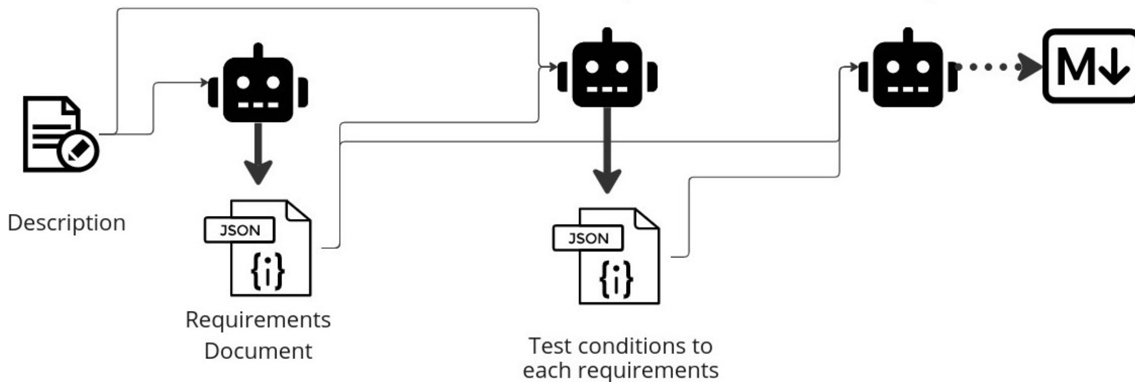

Figure 1. The workflow for building test cases using a Large Language Model (represented as the robot)

An interactive method was devised to derive test cases from the model, as outlined in Figure 1. Initially, engineers used a specially designed template to provide detailed application descriptions efficiently. This template, refined through extensive testing, helps pinpoint the most relevant information for the model to grasp the software context. The model then generates a requirements document, which, combined with the initial description, is used to formulate and produce test cases in the required format.

The OpenAI API was used with the GPT-3.5 Turbo model [OpenAI 2023] together with the Python framework LangChain [LangChain 2023] in order to manage the prompts, insert the context, memory, and handle interactivity. The following subsections details the step by step of this procedure.

3.1.1. Application Description

Providing contextual information is crucial for tailoring the test cases to the specific application at hand and preventing hallucinations, as emphasized by [Lewis et al. 2020]. Crafting this description should not become more challenging than generating the test cases themselves. To balance this trade-off and ensure accurate communication of relevant information, empirical experiments were conducted with various hypothetical software. These experiments involved a qualitative analysis to identify potential gaps or missing information. It was observed that using one template per software feature resulted in significantly better outcomes than attempting to describe all features in a single template file. Consequently, a template has been devised to guide the application description process:

What is the software name?

- A single line with the desired name to refer to the software

What is the software's main purpose?

- A brief description with no more than 3 lines

What is the platform type?

- The platform of the software: Web, mobile, ...

Feature description

- An overview of each feature from the software

3.1.2. Prompt Design

In the prompt elaboration, throughout the tests and references about prompt engineering, the input for the model was separated into three parts, with each one complementing the other, reducing the complexity and granting better quality in the responses [Gu et al. 2023]. The prompts use the concept of Role Prompting [Kong et al. 2023], which provides the model context in the way it should interpret the input, thus, contributing into more elaborated responses from the model.

To build the software requirements the prompt is consisted of the role description (Ex.: QA Engineer), the feature description as stated in section 3.1.1 with all information filled according to the under testing software, and the instructions for the expected output:

- `Project name`
- `Project goal`
- `List of the software's non-functional requirements.`
- `The list of the software's functional requirements.`

The first two attributes are used to confirm that the model has accurately absorbed the given context and can succinctly summarize the software's purpose. Additionally the instruction also request to format the output as JSON, enabling its seamless integration as input for subsequent prompts.

The next step is to produce the test conditions. This includes incorporating the role, the software description, and the generated software requirements from the previous step into the prompt. As part of the prompt's instructions, the desired output is specified to be a list of test conditions for each functional requirement, formatted in JSON.

In the final step, test cases are built for each test condition. This process involves looping over the list created in the previous step. Each prompt includes several key elements: the role, the software description, the requirement, and the test condition itself. For the output, markdown format is specified. This choice facilitates the subsequent conversion of the test cases into a PDF document, significantly improving their visualization and organization. The test cases should incorporate several essential attributes: `ID`, `Title`, `Pre-conditions`, `Steps`, `Expected results`, `Test Data`.

3.2. Evaluation setup

To effectively evaluate test case generation, we selected Da.tes, an active web platform with a broad range of features for connecting startups, investors, and businesses, as our

study subject. The platform's diverse functionalities facilitated a comprehensive analysis of test case generation effectiveness. Access to Da.tes' existing manual test cases provided a crucial comparison point for our research.

Our evaluation centered on the quality of test case documentation, sidestepping metrics that necessitate code access or historical data, focusing on qualities like Clarity, Correctness, Completeness, Consistency, Understandability, Self-containedness, and Specificity, as outlined in [Barraood et al. 2021a] and [Barraood et al. 2021b].

The study employed an unbiased approach, using anonymous, uniformly formatted sets of 10 AI-generated and 10 manual test cases, assessed by quality assurance engineers to gauge documentation quality effectively.

Two separate forms were created for the evaluation. The first form was designed to focus explicitly on the quality sub-factors of each test case under analysis. To mitigate volunteer fatigue and potential confusion, only the three most critical factors identified by [Lai 2017] were used: **Correctness**, **Completeness**, and **Consistency**. Volunteers were asked to rate each factor for each test case on a scale of 1-5, in accordance with the criteria outlined in Table1.

Score	1	2	3	4	5
Meaning	Very Poor	Poor	Regular	Good	Very Good

Table 1. Scores definition that was used by volunteers to rate the test cases

The second form focused on the implicit quality of the test cases, employing an A/B testing format to gauge preferences. In this setup, two similar test cases—each testing the same path in the software—were presented side by side. Volunteers were given 10 questions, asking them to choose between test A or test B, without knowing which was AI-generated and which was manually crafted. Additionally, a justification field was provided, where they could explain their choice. This feedback, inherently subjective and not based on predefined quality factors like the first form, aims to uncover any discernible differences between the two groups of test cases that may warrant further investigation

4. Results and Discussions

A total of 7 participants completed both forms. While the analysis is based on this limited sample, the consistency in responses and the insightful explanations submitted by participants offer relevant insights. We start our analysis with the quantitative scores for each metric. To provide an overview, the average scores for each group were calculated. Figure 2 shows the comparison between AI and Manual test cases. The total average scores across all three metrics are similar - 4.31 for AI versus 4.18 for Manual - with a slight advantage for the AI-generated process.

The correctness metric revealed the most significant discrepancy between the AI and Manual test groups in our evaluation. A closer examination of individual results indicated that AI-generated tests never received the lowest score. However, the manually written tests did receive the lowest score in a few cases, thereby lowering their average. Despite this, the overall assessment suggests that the AI is capable of generating test cases with a level of correctness comparable to human performance.

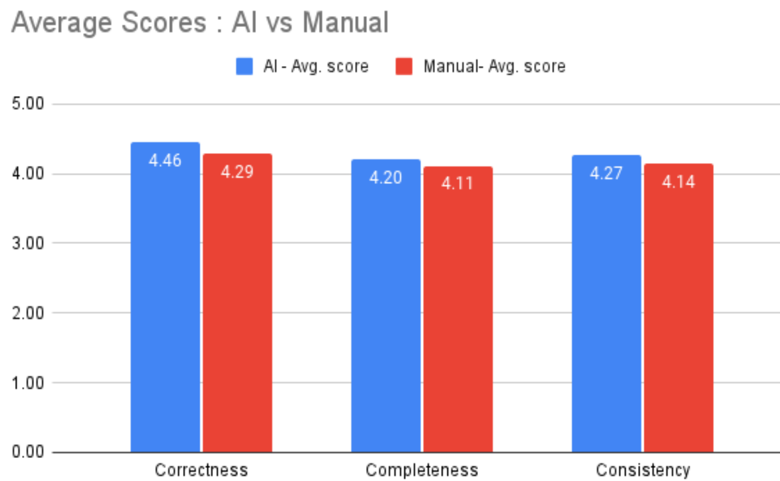

Figure 2. Average rate for AI and Manual test cases group split by quality factors

The same observation applies to the consistency of the test cases, where some instances of very low scores explain the observed gap. Consistency, defined as the degree of uniformity and reliability in test outcomes, reflects the ability of a set of test cases to produce reliable and expected results under varying conditions. In our analysis, it was evident that the test cases created by the AI maintained a consistent level of logic, even though they were generated in separate interactions. This consistency is likely attributable to the specific prompt that guided the output format, ensuring a uniform approach across different test cases.

In the analysis of completeness, we observed that the averages were quite similar between the two groups. However, there was a notable difference in their distributions. The manually created tests received more top scores (35 rated as '5') compared to the AI-generated tests (which received 32 top scores), but they also had a greater number of the lowest scores. Completeness, which refers to the extent to which a set of test cases covers all possible scenarios and functionalities of the software, can vary significantly. This variation in scores can be attributed to the test engineer's personal judgment, particularly in deciding the level of detail needed based on the perceived importance or rarity of a feature.

The second questionnaire provides more insight into the perceptions and preferences regarding the test cases, deliberately avoiding inference of any specific metrics, as it was presented in an A/B testing format. Given the total of the responses for all tests (70 - 7 evaluators and 10 test cases), the overall preference leaned towards the AI-generated cases (58.6%). These results are illustrated in Figure 3.

In analyzing the results on a case-by-case basis, we defined a 'clear win' as a scenario where the preference difference exceeds one vote (e.g., a 4 versus 3 vote count is considered a tie). Applying this criterion, we found that only two AI-generated tests were preferred over their manual counterparts, and similarly, two manual tests were favored over the AI ones. The remaining six test cases resulted in what can be termed as a technical tie.

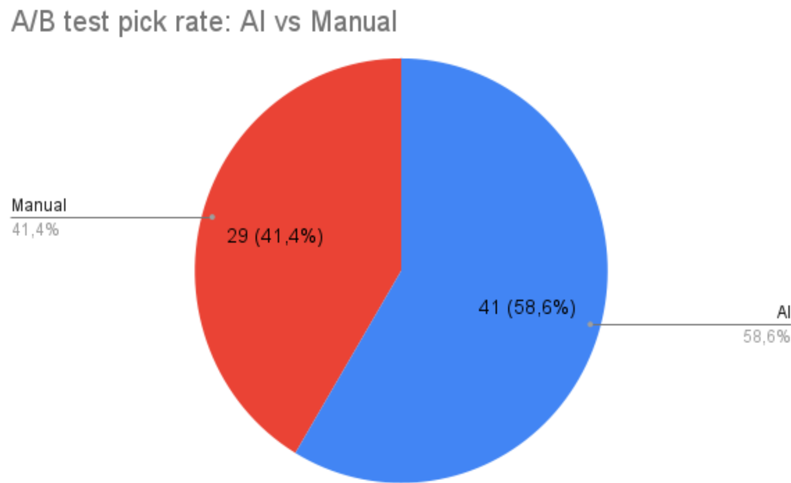

Figure 3. Pick rate for AI and Manual test cases in the A/B test

In delving into the reasons provided when Manual test cases were preferred, common themes emerged around correctness and completeness, with 'clarity' being a frequently cited new factor. Conversely, when AI-generated test cases were favored, the justifications predominantly revolved around the level of detail and completeness. The rationale behind the ties varied, ranging from perceptions of equivalency between the cases to differing opinions on correctness. For example, in the evaluation of the same test case, some evaluators identified a mistake, while others considered it more correct and complete.

Our analysis reveals a complex picture of AI-generated versus manually created test cases, highlighting the trade-offs and potential synergies between the two. Preferences vary, with evaluators considering factors like clarity, correctness, detail, and completeness, challenging the conventional views on automated versus manual testing. These insights, demonstrating the nuanced aspects of test case quality, pave the way for a deeper understanding of AI's role in quality assurance as we delve into the concluding discussions of this paper.

5. Conclusion

This paper explored the use of Large Language Models (LLMs) in generating test cases for software engineering, focusing on a case study with the software application Da.tes. We found that LLMs can create test cases that rival those made by humans in quality, showcasing their potential as tools in test case development, though human input remains crucial for ensuring accuracy and context-appropriateness. While LLMs enhance efficiency and detail in test creation, their effectiveness varies, indicating the need for a combined approach with human expertise. Future research should investigate the broader implications, including cost and time benefits, of integrating LLMs in test case generation. Despite some limitations in handling complex inter-feature tests, enhancing the LLM's understanding of feature interdependencies could significantly improve its applicability in more intricate testing scenarios.

References

Barraood, S., Mohd, H., and Baharom, F. (2021a). Test case quality factors. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12:1683–1694.

Barraood, S., Mohd, H., and Baharom, F. (2021b). Test case quality factors: Content analysis of software testing websites. *Webology*, 18:75 – 87.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Gu, J., Han, Z., Chen, S., Beirami, A., He, B., Zhang, G., Liao, R., Qin, Y., Tresp, V., and Torr, P. (2023). A systematic survey of prompt engineering on vision-language foundation models. *arXiv preprint arXiv:2307.12980*.

Kim, S., Zhao, J., Tian, Y., and Chandra, S. (2021). Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE.

Kong, A., Zhao, S., Chen, H., Li, Q., Qin, Y., Sun, R., and Zhou, X. (2023). Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702*.

Lai, S.-T. (2017). Test case quality management procedure for enhancing the efficiency of iid continuous testing. *Journal of Software*, 12(10):794–806.

LangChain (2023). Langchain: Building applications with llms through composability. https://github.com/langchain-ai/langchain.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

OpenAI (2023). Openai: Advancing digital intelligence. https://www.openai.com. Accessed on November 21, 2023.

Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S. K., and Sundaresan, N. (2021). Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models.