Automated Test Generation Using LLM Based on BDD: A Comparative Study

Shexmo Richarlison Ribeiro dos Santos[1][a], Luiz Felipe Cirqueira dos Santos[1][b], Marcus Vinicius Santana Silva[1][c], Marcos Cesar Barbosa dos Santos[1][d], Mariano Florencio Mendonça[1][e], Marcos Venicius Santos[1][f], Marckson Fábio da Silva Santos[1][g], Alberto Luciano de Souza Bastos[1][h], Sabrina Marczak[2][i], Michel S. Soares[1][j] and Fabio Gomes Rocha[3][k]

[1]*Federal University of Sergipe, São Cristóvão, Sergipe, Brazil*
[2]*School of Technology, PUCRS, Porto Alegre, Rio Grande do Sul, Brazil*
[3]*ISH (SafeLabs), Vitória, Espírito Santo, Brazil*

Keywords: Software Quality, Behavior-Driven Development (BDD), Large Language Models (LLM), Automatic Test Code Generator, Experiment.

Abstract: In Software Engineering, seeking methods that save time in product development and improve delivery quality is essential. BDD (Behavior-Driven Development) offers an approach that, through creating user stories and acceptance criteria in collaboration with stakeholders, aims to ensure quality through test automation, allowing the validation of criteria for product acceptance. The lack of test automation poses a problem, requiring manual work to validate acceptance. To solve the issue of test automation in BDD, we conducted an experiment using standardized prompts based on user stories and acceptance criteria written in Gherkin syntax, automatically generating tests in four Large Language Models (ChatGPT, Gemini, Grok, and GitHub Copilot). The experiment compared the following aspects: response similarity, test coverage concerning acceptance criteria, accuracy, efficiency in the time required to generate the tests, and clarity. The results showed that the LLMs have significant differences in their responses, even with similar prompts. We observed variations in test coverage and accuracy, with ChatGPT standing out in both cases. In terms of efficiency, related to time, Grok was the fastest while Gemini was the slowest. Finally, regarding the clarity of the responses, ChatGPT and GitHub Copilot were similar and more effective than the others. The results show that the LLMs adopted in the study can understand and generate automated tests accurately. However, they still do not eliminate the need for human assessment, but they do serve as a support to speed up the automation process.

1 INTRODUCTION

Behavior-Driven Development (BDD) is a framework often used in software development. Frameworks are standardized methods used in the software development process to contribute to understanding the logical and sequential steps involved, helping developers to understand the process as a whole.

BDD was created in 2003 by Dan North (North, 2006) to mitigate the issues arising from Test-Driven Development (TDD). While TDD focuses on testing the software, BDD aims to determine the behaviour the software needs to exhibit when executing a par-

[a] https://orcid.org/0000-0003-0287-8055
[b] https://orcid.org/0000-0003-4538-5410
[c] https://orcid.org/0009-0000-9211-5259
[d] https://orcid.org/0000-0002-7929-3904
[e] https://orcid.org/0000-0003-0732-3980
[f] https://orcid.org/0009-0006-1645-6127
[g] https://orcid.org/0009-0001-6479-1900
[h] https://orcid.org/0009-0002-3911-9757
[i] https://orcid.org/0000-0001-9631-8969
[j] https://orcid.org/0000-0002-7193-5087
[k] https://orcid.org/0000-0002-0512-5406

Ribeiro dos Santos, S. R., Cirqueira dos Santos, L. F., Silva, M. V. S., Barbosa dos Santos, M. C., Mendonça, M. F., Santos, M. V., Santos, M. F. S., Bastos, A. L. S., Marczak, S., Soares, M. S. and Rocha, F. G.
Automated Test Generation Using LLM Based on BDD: A Comparative Study.
DOI: 10.5220/0013683600003985
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 21st International Conference on Web Information Systems and Technologies (WEBIST 2025)*, pages 47-58
ISBN: 978-989-758-772-6; ISSN: 2184-3252
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

ticular functionality. Thus, BDD is used throughout the system lifecycle, from requirements elicitation to automation and validation (Bruschi et al., 2019).

Requirements elicitation using BDD uses the "given, when, then" pattern, employing an easily understandable language known as *Gherkin* (Smart, 2014) to enhance stakeholder communication and understanding. Through this pattern, the software components' requirements are elicited, focusing on outlining the expected behaviour of the software.

The use of generative Artificial Intelligence (AI) in software development has been evident in various tools that enhance productivity, provided they are adequately supervised (Sauvola et al., 2024). Therefore, it is natural to employ generative AI tools for testing and documentation purposes to optimize BDD processes.

Therefore, the main goal of this research is to "**Analyze** the efficiency of user stories using BDD, **aiming at** automatic generation of test code, **in comparison to** Large Language Models (LLMs), **from an** academic **perspective, in the context of** software development". To achieve this goal, the posed the following research questions.

RQ1. What is the similarity of responses among the different AIs when generating automated tests?

RQ2. What is the coverage of acceptance criteria by the tests generated by each AI?

RQ3. What is the accuracy of the generated tests compared to a reference test set?

RQ4. How much time is required to generate the tests by each LLM?

RQ5. What is the clarity of responses among different executions of each AI?

To answer them, we conducted an experiment, where a software was created to read user stories and their respective acceptance criteria, generating test code using different AIs. Using the prompt we created, we were able to insert them into the AIs used so that we could observe what the return would be from them. In this way, we were able to analyze whether the use of AIs is effective in this context.

Our paper brings two main contributions: the demonstration that LLMs can be used to generated automated tests based on user stories and acceptance scenarios and the comparison among the performance of the selected LLMs: ChatGPT, Gemini, Grok, and GitHub Copilot, for each one of the investigated aspects.

The remainder of this paper is organized as follows: Section 2 explains the concepts inherent to the use of BDD and AI. Section 3 presents related work. Section 4 details our experiment procedures and methods. Section 5 reports our study results and Section 6 discusses them with regards literature. Section 7 presents the threats to the validity of our study. Section 8 concludes the paper by highlighting once again its contributions and presenting proposed future work.

2 BACKGROUND

Next, concepts inherent to Behavior-Driven Development (BDD) and Large Language Model (LLM) are discussed, including the presentation of the frameworks used in this research for the automatic generation of test code.

2.1 Behavior-Driven Development

The Agile Movement (Beck et al., 2001) gathered computing experts to seek improvements in software quality and time response of software delivery. From this point, changes occurred, and frameworks were created to enable faster deliveries. Created after this Movement, Behavior-Driven Development (BDD) is an agile framework used throughout the software development cycle.

Created by Dan North (North, 2006), BDD aims to improve communication among those involved in software development, specially communication with stakeholders who are often not familiar with technical language, enhancing the quality of software delivery as a consequence. Thus, some benefits of BDD include time optimization and enhanced quality in requirements elicitation (Pereira et al., 2018).

BDD is characterized by the use of the *Gherkin* language (Smart, 2014), written in a clear, direct, and assertive natural language, contributing to high-quality requirements elicitation to find the expected behaviour of software, i.e., the stakeholder's needs. BDD divides the writing of requirements into two parts: the first being the functionality or expected behaviour of the system, and the second part being one or more acceptance criteria related to the validation of this behaviour (North et al., 2019), as illustrated in Figure 1.

BDD uses the terms "given, when, then" as a standard writing format, with each user story having its respective acceptance criteria. Thus, the scenarios or criteria are written to be testable (Silva and Fitzgerald, 2021), aiding in the specification and verification of requirements (Guerra-Garcia et al., 2023).

Since its functionalities are written concisely, user stories are easily understood by all involved, improving communication among stakeholders (Couto et al., 2022; Pereira et al., 2018; Bruschi et al., 2019).

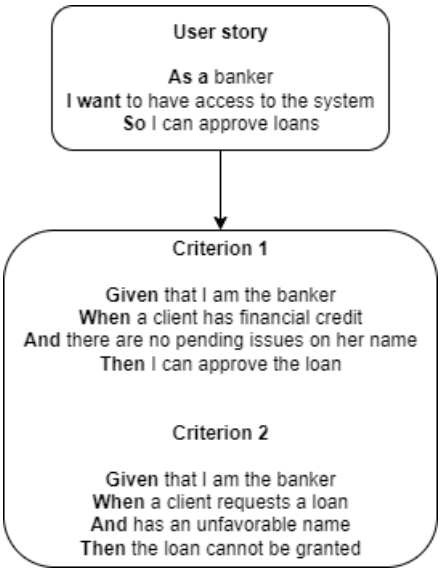

Figure 1: User story example.

By enhancing communication, the *Gherkin* language adopted by BDD ensures assertiveness in the software behaviour, making all parties aware of what a particular user story refers to, for example.

However, BDD presents specific challenges, among which the gap related to the misalignment between acceptance criteria and automation stands out. Implementing tests is a manual task, requiring a systematic process for efficient integration (Zameni et al., 2023). The lack of precision and coverage of the tests concerning the acceptance criteria can result in incomplete validation (Zameni et al., 2023). In addition, the manual creation of tests increases the workload, emphasising the need for systematic flows and support tools to effectively integrate tests in the BDD context (Ma et al., 2023).

Although BDD has been used successfully in various software engineering processes, studies still need to explore the enhancement of BDD through emerging technologies such as machine learning (Binamungu and Maro, 2023).

2.2 Large Language Model

Language Models refer to any system trained to predict a series of characters, whether letters, words, or sentences, sequentially or not, given some previous or adjacent context (Bender and Koller, 2020). The development of Language Models follows two main approaches: transformer models and word embeddings.

Word embeddings improve the results of various performance tests while reducing the labelled data needed for multiple supervised tasks. In contrast, transformer models have continuously benefited from larger architectures and datasets, with their capacity subsequently enhanced for specific tasks. Some of these models have redefined the concept of their classification, making it more accurate to characterize them as Large Language Models (LLMs) (Bender et al., 2021), such as GPT (OpenAI), Gemini (Google), Grok (Twitter), and GitHub Copilot.

Recent studies indicate that using LLMs can reduce the time required to write and maintain BDD tests and improve the quality and coverage of tests (Zhang et al., 2023). This methodology smooths the barriers between requirements gathering and technical implementation, allowing for more effective collaboration between stakeholders and the development team, and promoting a more agile and integrated software development cycle.

3 RELATED WORK

Large language models (LLMs) are increasingly used in Software Engineering (SE) for different tasks, like generating code, designing software, and automating test cases. As a result, in this study, we want to highlight related work identified in Table 1. We will present this information next.

The principle of using natural language to create high-level code with AI is addressed by Lee *et al.* (Lee et al., 2023), using the OpenCV tool for a UI test in an object-oriented analysis combined with BDD for a test automation. They experiment with natural language-based templates and then apply it to a UI test simulator. The results were analyzed using a BDD and an OOBDD (object-oriented behavior-driven design) approach. Through their results, we understood how generative AI works efficiently for human-language test automation for complex software.

In contrast, Takerngsaksiri *et al.* (Takerngsaksiri et al., 2024) share the results of using PyTester, a Text-to-TextCase tool in a TDD scenario. They compare results from PyTester with the state-of-art models directly (Finetuned CodeT5-large, Incoder,Starcoder e GPT3.5). The results reinforce how the use of AI-automated tools based on natural language are efficient with a low percentage of errors or inconsistencies.

Mock *et al.* (Mock et al., 2024) analyses the interaction between the development team without AI-assist and the test codes generated by AI, identifying which ones are the most promising. With the aim of automating TDD processes with artificial intelligence, the study compared the generated code with the other 5 developers, concluding that the automa-

Table 1: Related Work.

Reference	Abstract
(Lee et al., 2023)	Demonstrates an approach using generative AI to translate human language into high-level programming language.
(Mock et al., 2024)	Introduces an approach that proposes the automation of TDD through Generative AI.
(Karpurapu et al., 2024)	Evaluates a detailed approach focusing on enhancing BDD practices through LLMs to generate acceptance tests automatically.
(Takerngsaksiri et al., 2024)	Presents and evaluates PyTester as a tool for generating formal test cases from natural language.

tion of the TDD process can indeed be used efficiently but with due supervision due to the quality of the code that is produced in this way.

Regarding a multi-AI analysis, Karpurapu *et al.* (Karpurapu et al., 2024) concludes that BDD acceptance tests generated by LLMs are beneficial, as these tests represent considerable complexity.

The use of Large Language Models (LLMs) for automated code generation following Test-Driven Development (TDD) and Behavior-Driven Development (BDD) processes has shown satisfactory and promising results, warranting further exploration and expansion. Our article builds on this foundation by using code generated from generative AI. However, we take it a step further by introducing a novel approach of comparing the outputs from various LLMs using the prompt and evaluating them through specific metrics, which will be presented throughout the article.

4 METHODOLOGY

Experiments are an empirical method that aids in the evaluation and validation of research results (Wohlin et al., 2003).In Software Engineering, experiments aim to identify the outcomes of certain situations and seek to benefit the field with potential discoveries.

For this experiment, a library provided by a company containing 34 user stories was used. Of these, 30 stories had three acceptance criteria, and four had only one criterion, adding 94 acceptance criteria. The stories were written in *Gherkin* language using the BDD framework in the native language of the researchers (Brazilian Portuguese). Based on these, a prompt was created for each scenario in the selected Large Language Models. The stories and their respective scenarios and prompts can be viewed at the link[1].

The objective at this point is to compare the effectiveness of LLMs Grok, Gemini, ChatGPT, and GitHub Copilot in generating automated tests based on user stories and acceptance criteria, following BDD in the *Gherkin* standard, compared to computerised tests performed by a development team. This objective can be divided into five sub-objectives: each part is related to one of the research questions and has their own metrics, and hypotheses detailed as follows:

Objective 1: Measure the similarity of responses from different LLMs

- **RQ1** - Question: What is the similarity of responses among the different AIs when generating automated tests?
- *Metric:* Similarity coefficient (*Cosine Similarity*).
- Null Hypothesis (H0.1): There is no significant difference in the similarity of responses among the different AIs.
- Alternative Hypothesis (H1.1): There is a significant difference in the similarity of responses among the different AIs.

Objective 2: Validate whether the results generated by the LLMs cover the acceptance criteria.

- **RQ2** - Question: What is the coverage of acceptance criteria by the tests generated by each AI?
- *Metric:* Acceptance criteria coverage (percentage of acceptance criteria covered by the generated tests).
- Null Hypothesis (H0.2): There is no significant difference in the coverage of acceptance criteria among the different AIs.
- Alternative Hypothesis (H1.2): There is a significant difference in the coverage of acceptance criteria among the different AIs.

Objective 3: Evaluate the accuracy of the tests generated by the different LLMs

[1]https://doi.org/10.5281/zenodo.13155965

- **RQ3** - Question: What is the accuracy of the generated tests compared to a reference test set?
- *Metric:* Test accuracy (percentage of correspondence between the generated tests and the reference test set).
- Null Hypothesis (H0.3): There is no significant difference in the accuracy of the tests generated among the different AIs.
- Alternative Hypothesis (H1.3): There is a significant difference in the accuracy of the tests generated among the different AIs.

Objective 4: Evaluate the efficiency, in terms of time, for generating the tests

- **RQ4** - Question: How much time is required to generate the tests by each LLM?
- *Metric:* Test generation time (average time required to generate tests).
- Null Hypothesis (H0.4): There is no significant difference in the test generation time among the different AIs.
- Alternative Hypothesis (H1.4): There is a significant difference in the test generation time among the different AIs.

Objective 5: Evaluate the clarity of responses among different executions of each AI

- **RQ5** - Question: What is the clarity of responses among different executions of each AI?
- *Metric:* Clarity of responses (evaluated by subjective criteria such as readability, comprehensibility, and adherence to acceptance criteria).
- Null Hypothesis (H0.5): There is no significant difference in the clarity of responses among the different AIs.
- Alternative Hypothesis (H1.5): There is a significant difference in the clarity of responses among the different AIs.

4.1 Experiment Execution

The methodological steps taken for executing this experiment are outlined next:

1. Submit each user story and their respective acceptance criteria to the LLMs Grok, Gemini, ChatGPT, and GitHub Copilot using a standard prompt;
2. Generate and document the test code returned by each source;
3. Execute the generated tests and record the results;
4. Statistically evaluate the results.

Table 2: Evaluated Metrics.

Metric	Definition	Data Collection
Accuracy	It refers to the proportion of tests that passed (correct results) among all executed tests.	After executing the generated tests, the number of tests that passed and failed was recorded.
Coverage	It refers to the proportion of requirements or acceptance criteria covered by the generated tests.	The number of acceptance criteria covered by the generated tests was checked for each user story.
Clarity	It refers to the readability and comprehension of the generated tests. It can be qualitatively evaluated by a group of developers or through automatic readability metrics.	Developers could assign a score from 1 to 5 for each generated test. Alternatively, readability metrics such as *Flesch Reading Ease* could be used.
Efficiency	It refers, in the context of this paper, to the time required to generate the tests.	The time from the test request to its generation and recording was measured.

4.2 Evaluated Metrics

As shown in Table 2, the metrics adopted in this study are partially related to the need to improve alignment between acceptance criteria and test automation. The acceptance criteria stage must have a positive result for the automation of its respective tests to be carried out. Thus, they must be in line with the behavior expected by the software. The metrics used were selected with the aim of achieving the objectives proposed in this study. In addition, the efficiency measured is associated with the time needed to create the tests, thus reducing the manual workload, as identified as a gap in the literature review.

Through these metrics, results can be more assertive, increasing the reliability of results and providing clarity and effectiveness to the conclusions of this research.

5 RESULTS

The results are related to each sub-objectives outlined in section 4, that refer to their respective research questions. The following are the results related to each of the objectives outlined in Section 4 with the aim of answering their respective RQs outlined in Section 1. The results are presented in Figures 2 and 3.

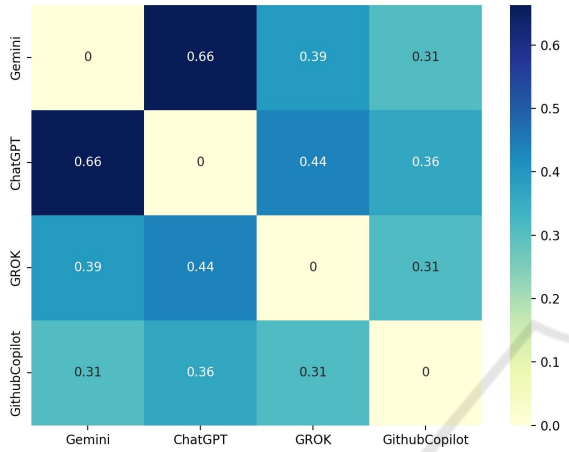

Figure 2: Similarity Matrix.

In Figure 3, “A” refers to LLM Gemini, “B” refers to ChatGPT, “C” refers to GROK, and “D” refers to GitHub Copilot.

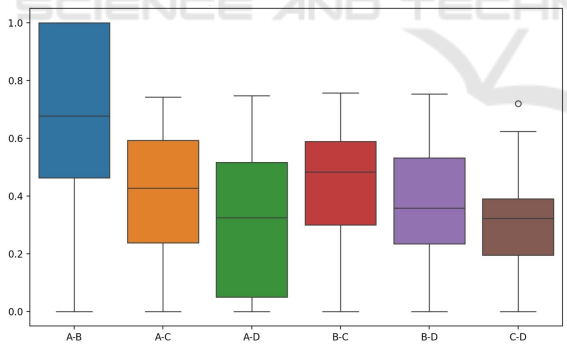

Figure 3: Distribution of similarities.

5.1 Objective 1

To measure the similarity of responses from different LLMs, the Kruskal-Wallis test was employed. This non-parametric test is appropriate for comparing independent distributions when the assumptions of normality are not met.

Results of the Kruskal-Wallis Test:

- Statistic: 36.2464
- p-Value: 0.0000

The results indicated a Kruskal-Wallis statistic of 36.2464 and a p-value of 0.0000. The extremely low p-value (less than 0.05) suggests that there is a statistically significant difference in the similarity of responses among the different LLMs.

Hypotheses:

- Null Hypothesis (H0.1): There is no significant difference in the similarity of responses among the different AIs.
- Alternative Hypothesis (H1.1): There is a significant difference in the similarity of responses among the different AIs.

Answering RQ1: Therefore, since the p-value is less than 0.05, we reject the null hypothesis (H0.1). Thus, we support the alternative hypothesis (H1.1), which asserts that there is a significant difference in the similarity of responses among the different LLMs. This implies that the LLMs Grok, Gemini, ChatGPT, and GitHub Copilot produce responses with statistically significant varying levels of similarity when generating automated tests based on user stories and acceptance criteria.

5.2 Objective 2

To validate if the results generated by the LLMs cover the acceptance criteria, a coverage analysis and an ANOVA test were conducted, followed by*post-hoc* and Tukey HSD.

Coverage Means:

- Grok: 0.4054
- Gemini: 0.5943
- ChatGPT: 0.7670
- GitHub Copilot: 0.7315

Coverage ANOVA:

- Sum of Squares (Model): 7.419452
- Sum of Squares (Residual): 13.830678
- F-Value: 65.089134
- p-Value: 1.025147e-33

The ANOVA results indicated an F-value of 65.089134 and a p-value of 1.025147e-33, which is extremely low (less than 0.05). This suggests that there is a statistically significant difference in the coverage of acceptance criteria among the different LLMs.

Tukey HSD Test for Coverage:

- ChatGPT vs. GitHub Copilot: p = 0.6063 (not significant)
- ChatGPT vs. Gemini: p ¡ 0.001 (significant)

- ChatGPT vs. Grok: p ¡ 0.001 (significant)
- GitHub Copilot vs. Gemini: p ¡ 0.001 (significant)
- GitHub Copilot vs. Grok: p ¡ 0.001 (significant)
- Gemini vs. Grok: p ¡ 0.001 (significant)

The results of the *post-hoc* Tukey HSD test showed that the differences in coverage of acceptance criteria are significant among most LLMs, except for ChatGPT and GitHub Copilot, whose differences were not significant (p = 0.6063).

Hypotheses:

- Null Hypothesis (H0.2): There is no significant difference in the coverage of acceptance criteria among the different AIs.
- Alternative Hypothesis (H1.2): There is a significant difference in the coverage of acceptance criteria among the different AIs.

Answering RQ2: Therefore, since the p-value of the ANOVA is less than 0.05, we reject the null hypothesis (H0.2). Thus, we support the alternative hypothesis (H1.2), which states that there is a significant difference in the coverage of acceptance criteria among the different AIs. The *post-hoc* tests indicate that, although ChatGPT and GitHub Copilot do not show significant differences between each other, all other comparisons between the LLMs are significantly different.

5.3 Objective 3

To assess the accuracy of tests generated by different LLMs, precision means were calculated and an analysis of variance (ANOVA) was conducted, followed by the Tukey HSD *post-hoc* test.

Precision Means:

- Grok: 0.3391
- Gemini: 0.5373
- ChatGPT: 0.7670
- GitHub Copilot: 0.7239

ANOVA of Precision:

- Sum of Squares (Model): 10.57519
- Sum of Squares (Residual): 12.50989
- F-Value: 102.56869
- P-Value: 3.882251e-48

The results of the ANOVA indicated an F-value of 102.56869 and a p-value of 3.882251e-48, which is extremely low (less than 0.05). This suggests that there is a statistically significant difference in the accuracy of tests generated by the different LLMs.

Tukey HSD Test of Precision:

- ChatGPT vs. GitHub Copilot: p = 0.3944 (not significant)
- ChatGPT vs. Gemini: p ¡ 0.001 (significant)
- ChatGPT vs. Grok: p ¡ 0.001 (significant)
- GitHub Copilot vs. Gemini: p ¡ 0.001 (significant)
- GitHub Copilot vs. Grok: p ¡ 0.001 (significant)
- Gemini vs. Grok: p ¡ 0.001 (significant)

The results of the Tukey HSD *post-hoc* test showed that the differences in test accuracy are significant between ChatGPT, Gemini, and Grok. There were no significant differences in accuracy between GitHub Copilot and ChatGPT (p = 0.3944).

Hypotheses:

- Null Hypothesis (H0.3): There is no significant difference in the accuracy of tests generated among the different AIs
- Alternative Hypothesis (H1.3): There is a significant difference in the accuracy of tests generated among the different AIs.

Answering RQ3: Given that the p-value of the ANOVA is less than 0.05, we reject the null hypothesis (H0.3). Therefore, we support the alternative hypothesis (H1.3), which states a significant difference in the accuracy of tests generated by the different LLMs. The results of the *post-hoc* test indicate that ChatGPT has a significantly different accuracy compared to the other LLMs tested, except for GitHub Copilot. At the same time, the differences between GitHub Copilot, Gemini, and Grok are also statistically significant.

5.4 Objective 4

To assess the efficiency of the different LLMs in test generation, the mean generation times were calculated and an analysis of variance (ANOVA) was conducted, followed by the Tukey HSD *post-hoc* test.

ANOVA of Efficiency:

- Sum of Squares (Model): 0.712254
- Sum of Squares (Residual): 0.028181
- F-Value: 3066.558824
- P-Value: 6.633292e-258

The ANOVA results indicate a significant difference between the groups, as the p-value is extremely low (6.633292e-258). This means that at least one of the models has a significantly different efficiency performance compared to the others.

Tukey HSD Test of Efficiency:

- ChatGPT vs. GitHub Copilot: p = 1.0 (not significant)
- ChatGPT vs. Gemini: p ¡ 0.001 (significant)
- ChatGPT vs. Grok: p = 0.1858 (not significant)
- GitHub Copilot vs. Gemini: p ¡ 0.001 (significant)
- GitHub Copilot vs. Grok: p = 0.1858 (not significant)
- Gemini vs. Grok: p ¡ 0.001 (significant)

Significant Differences:

- ChatGPT vs. Gemini: The difference is significant, indicating that the generation time of ChatGPT is significantly shorter than that of Gemini.
- GitHub Copilot vs. Gemini: The difference is significant, indicating that the generation time of GitHub Copilot is significantly shorter than that of Gemini.
- Gemini vs. Grok: The difference is significant, indicating that the generation time of Gemini is significantly longer than that of Grok.

Non-Significant Differences:

- ChatGPT vs. GitHub Copilot: There is no significant difference, suggesting that the generation time of ChatGPT is similar to that of GitHub Copilot.
- ChatGPT vs. Grok: There is no significant difference, indicating that the generation time of ChatGPT is similar to that of Grok.
- GitHub Copilot vs. Grok: There is no significant difference, indicating that the generation time of GitHub Copilot is similar to that of Grok.

Interpretation:

- ChatGPT e GitHub Copilot: Both have comparable and efficient test generation times, with no significant differences between them.
- Gemini: The Gemini model exhibits significantly longer generation times compared to all other models, indicating inefficiency in its test generation process.
- Grok: The Grok model performs efficiently, similar to ChatGPT and GitHub Copilot, and significantly better than Gemini.

The results suggest that, in terms of test generation time efficiency, both ChatGPT and GitHub Copilot are effective and comparable. However, the Gemini model is significantly slower, indicating that it may not be the best choice when generation time is a critical factor. The Grok model also demonstrates efficiency and is comparable to ChatGPT and GitHub Copilot.

Hypotheses:

- Null Hypothesis (H0.4): There is no significant difference in the test generation time among the different AIs.
- Alternative Hypothesis (H1.4): There is a significant difference in the test generation time among the different AIs.

Answering RQ4: Given that the p-value of the ANOVA is less than 0.05, we reject the null hypothesis (H0.4). Therefore, we support the alternative hypothesis (H1.4), which states that there is a significant difference in the test generation time among the different AIs.

5.5 Objective 5

To assess the clarity of responses across different executions of each AI, clarity means were calculated and an analysis of variance (ANOVA) was conducted, followed by the Tukey HSD *post-hoc* test.

ANOVA of Clarity:

- Sum of Squares (Model): 145030.766421
- Sum of Squares (Residual): 124106.926135
- F-Value: 141.789559
- P-Value: 7.339233e-61

The results of the ANOVA indicate a significant difference in the clarity of responses among the different AIs, as the p-value is extremely low (7.339233e-61). This means that at least one of the AIs has significantly different clarity than the others.

Means of Clarity:

- Grok: 44.6511
- Gemini: 67.0604
- ChatGPT: 92.2609
- GitHub Copilot: 92.2609

Tukey HSD Test of Clarity:

- ChatGPT vs. GitHub Copilot: p = 1.0 (not significant)
- ChatGPT vs. Gemini: p ¡ 0.001 (significant)
- ChatGPT vs. Grok: p ¡ 0.001 (significant)
- GitHub Copilot vs. Gemini: p ¡ 0.001 (significant)
- GitHub Copilot vs. Grok: p ¡ 0.001 (significant)
- Gemini vs. Grok: p ¡ 0.001 (significant)

Significant Differences:

- ChatGPT vs. Gemini: The difference is significant, indicating that the clarity of responses from ChatGPT is significantly higher than that of Gemini.

- ChatGPT vs. Grok: The difference is significant, indicating that the clarity of responses from ChatGPT is significantly higher than that of Grok.
- GitHub Copilot vs. Gemini: The difference is significant, indicating that the clarity of responses from GitHub Copilot is significantly higher than that of Gemini.
- GitHub Copilot vs. Grok: The difference is significant, indicating that the clarity of responses from GitHub Copilot is significantly higher than that of Grok.
- Gemini vs. Grok: The difference is significant, indicating that the clarity of responses from Gemini is significantly higher than that of Grok.

Non-Significant Differences:

- ChatGPT vs. GitHub Copilot: There is no significant difference, suggesting that the clarity of responses from ChatGPT is similar to that of GitHub Copilot.

Interpretation

- ChatGPT e GitHub Copilot: Both AIs have responses with comparable clarity and significantly higher than the other AIs evaluated.
- Gemini: The Gemini model exhibits intermediate response clarity, being significantly better than Grok but worse than ChatGPT and GitHub Copilot.
- Grok: The Grok model has the lowest response clarity among all evaluated AIs.

Thus, the results suggest that ChatGPT and GitHub Copilot are adequate and comparable in terms of response clarity. The Gemini model is intermediate, exhibiting significantly higher clarity than Grok but lower than ChatGPT and GitHub Copilot. The Grok model shows the lowest clarity among all AIs.

Hypotheses:

- Null Hypothesis (H0.5): There is no significant difference in the clarity of responses among the different AIs.
- Alternative Hypothesis (H1.5): There is a significant difference in the clarity of responses among the different AIs.

Answering RQ5: Since the p-value of the ANOVA is less than 0.05, we reject the null hypothesis (H0.5). Therefore, we support the alternative hypothesis (H1.5), which states a significant difference in the clarity of responses among the different AIs.

6 DISCUSSION

Figure 4 was created using the model developed by Rajbhoj *et al.* (Rajbhoj et al., 2024). The adaptation created for this study made it possible to follow a step-by-step method inherent to executing the automatic test generator created here.

As can be seen, there is initially a stakeholder who has a desire for a particular behaviour performed by the software. Thus, the user story initially emerges, which, in turn, leads to one or more acceptance criteria, scenarios testable that can guarantee the return desired by the software. From this point on, the programming language followed by the testing framework is selected, intending the standard prompt to be used in LLMs can be created, and finally, the automatic code can be generated.

Furthermore, this study allowed reviewing an integrated cycle for using LLMs associated with the BDD points. The LLMs are incorporated as feedback points after formalizing the user story and its acceptance criteria. During development, LLMs may be asked to review and improve the source code during the TDD refactoring stage. This cycle is illustrated in Figure 5.

In addition to the previous cycle, the work began with the formalization of user stories and their respective acceptance scenarios provided by the company. These stories and scenarios were integrated into the prompt presented in Figure 6 prompt, making it possible to carry out the necessary tests in the LLMs Grok, Gemini, ChatGPT and GitHub Copilot, using the Python programming language to observe the similarity resulting from each one. Therefore, during the development cycle, after test generation, we implement a new interaction that can benefit the development process by using LLM, allowing LLM feedback during the TDD refactoring stage.

The model generates the code. Otherwise, as TDD and BDD themselves highlight refactoring as a crucial factor, a cycle involving new requests to LLM may be necessary, automating the development, execution and refactoring process with the support of LLM.

Regarding accuracy, ChatGPT and GitHub Copilot performed the best, being very close to each other. This result is because GitHub Copilot uses parts of the ChatGPT model. On the other hand, Gemini and Grok had significantly lower accuracies, suggesting that the different models were more effective, in part due to the use of the free version but also because, in general, Gemini has difficulty delivering tests for three scenarios in one single command. Therefore, we suggest submitting one scenario at a time during refactoring.

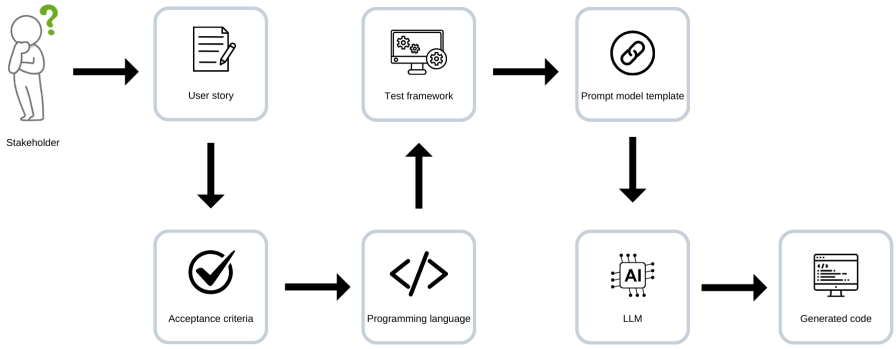

Figure 4: Prompt model for BDD test automation.

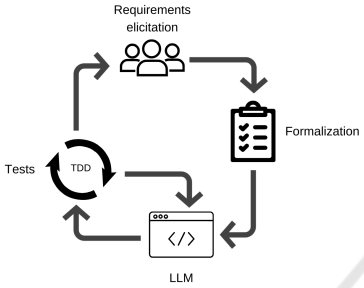

Figure 5: Automatic code generation.

As for clarity, Grok had the worst performance, mainly because it often generates results in English, which is in line with the platform's main focus being the English language. Another point to consider is that Grok focuses more on conversation and research based on data from *Twitter*, not having code generation as its primary objective. In the free version, Gemini presents good clarity when analyzing a single acceptance criterion but has difficulty generating code for multiple scenarios. There is a need to evaluate the advanced version to see if this issue is resolved.

The results showed that ChatGPT and the development team were effective and comparable in terms of test generation time efficiency. The Gemini model, however, was significantly slower, indicating that it may not be the best choice when generation time is a critical factor. The Grok model proved efficient and comparable to ChatGPT and the development team.

We observed that low-quality stories and scenarios negatively impact automatic code generation. This occurs due to ambiguities or unclear texts, making it difficult for LLMs to read and causing confusion in delivering the expected text. BDD was developed to be objective and concise; however, if a scenario is prepared with poor-quality writing, the return will certainly not be as expected.

The correct use of BDD and an LLM can benefit software development by helping developers automate test code. However, it is essential to emphasize that using this technology does not mean replacing the professional with the machine but instead taking advantage of existing technologies to assist in the necessary work.

We contributed to the advancement of the studies presented in Section 3 by carrying out a comparative experiment using the LLMs Gemini, Grok, ChatGPT and GitHub Copilot together with the BDD framework, demonstrating the effectiveness of well-written user stories with the objective of automatically generate test codes through AI.

7 THREATS TO VALIDITY

Threats to validity are understood as circumstances encountered during the study's execution, which need to be explained as they were mitigated, bringing reliability to the research (Runeson and Höst, 2009). Below, the threats encountered in this study are described according to Zhou *et al.* (Zhou et al., 2016).

7.1 Construction Validity

To conduct this study, a theoretical approach to BDD was necessary to understand how the framework functions and information about the LLMs used in this research. These details are provided in Section 2, where two authors addressed the critical concepts related to the themes of this study, synthesizing essential information for understanding the achieved results.

7.2 Internal Validity

Seven researchers conducted the study: three conducted the relevant theoretical research on the topic, one created the test codes, one performed the statistical analyses and supervised the paper, and two reviewed the study for improvements in the quality of

```
You are a test analyst, responsible for creating unit tests,
    based on the user story {us} and the acceptance criteria {ac}
    based on gherkin. Create unit tests for the programming
    language {pl} based on the {fw} framework and explain how to
    use the code.
```

Figure 6: Prompt for the Tests Generation.

the final product. All authors read and approved the final version of the article. There were no objections from any of the authors.

7.3 External Validity

To ensure the reliability of the study, user stories, their respective acceptance criteria, and prompts were created to be used with the selected LLMs, resulting in the generation of automated test code. These data can be viewed at this link[2].

7.4 Conclusion Validity

The step-by-step process of this research was described in Section 4, and the findings are presented in Section 5. This study is replicable if the methodological steps used are followed.

8 CONCLUSION

This work adopted an experimental research strategy using statistical evaluation based on 34 user stories and a total of 94 acceptance scenarios to analyze the similarity between the responses, the coverage of the tests generated to the indicated scenarios, the accuracy of the tests, the efficiency to generation time and, finally, the clarity of responses.

Creating automated tests using Large Language Models (LLMs) through Behavior-Driven Development (BDD) has proven to be a relevant approach in software development. However, we have identified that faster LLMs currently do not provide satisfactory results in clarity and accuracy, which suggests that speed should not be the main criterion when choosing an LLM.

The LLMs used in this study could understand and generate natural language text with precision and quality based on well-described user stories and acceptance scenarios. This aspect allows software engineers and quality assurance teams to automate the creation of their tests based on BDD acceptance scenarios, using natural language descriptions, speeding up development and ensuring that tests more accurately reflect the precise requirements of the business.

Although the results have shown promise, we are still far from complete automation that would allow human evaluation to be dispensed with LLMs speed up the initial creation steps, improving quality and saving time. Still, the scenarios must be written of high quality so that the resulting codes achieve the expected software behaviour. Good scenario writing is crucial to ensuring that automated test codes are effective.

The comparative tests brought reliability to this study, demonstrating that, as the software was created to interpret the stories with the LLMs, the tests could show whether the automated codes were being generated in a meaningful way based on the hypotheses presented.

One way to improve LLMs' responses for future work could be to implement the checklist presented by Oliveira, Marczak, and Moralles (Oliveira et al., 2019) in creating user stories with BDD. This could contribute to delivering more accurate test codes through the AutoDevSuite tool developed in this study.

ACKNOWLEDGMENTS

Sabrina Marczak would like to thank CNPq for the financial support (Productivity Scholarship, process no. 313181/2021-7). Shexmo Santos would like to thank CAPES/Brazil for the financial support (Master's Scholarship, process no. 88887.888613/2023-00).

REFERENCES

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). The agile manifesto.

Bender, E., Gebru, T., McMillan-Major, A., and Shmitchell, S. (2021). On the dangers of stochastic parrots: Can

[2]https://doi.org/10.5281/zenodo.13155965

language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency, FAccT '21*, pages 610–51998623. ACM.

Bender, E. and Koller, A. (2020). Climbing towards nlu: On meaning, form, andunderstanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–51998. ACL.

Binamungu, L. P. and Maro, S. (2023). Behaviour driven development: a systematic mapping study. *Journal of Systems and Software*, 203:111749.

Bruschi, S., Xiao, L., Kavatkar, M., et al. (2019). Behavior Driven-Development (BDD): a case study in healthtech. In *Pacific NW Software Quality Conference*.

Couto, T., dos Santos Marczak, S., Callegari, D. A., Móra, M., and Rocha, F. (2022). On the Characterization of Behavior-Driven Development Adoption Benefits: A Multiple Case Study of Novice Software Teams. *Anais do XXI Simpósio Brasileiro de Qualidade de Software, 2022, Brasil.*

Guerra-Garcia, C., Nikiforova, A., Jiménez, S., Perez-Gonzalez, H. G., Ramírez-Torres, M. T., and Ontañon-García, L. (2023). ISO/IEC 25012 - Based methodology for managing data quality requirements in the development of information systems: Towards Data Quality by Design . *Data and Knowledge Engineering*, 145:102152–102152.

Karpurapu, S., Myneni, S., Nettur, U., Gajja, L. S., Burke, D., Stiehm, T., and Payne, J. (2024). Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation. *IEEE Access*.

Lee, E., Gong, J., and Cao, Q. (2023). Object oriented bdd and executable human-language module specification. In *2023 26th ACIS International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD-Winter)*, pages 127–133. IEEE.

Ma, S.-P., Chen, Y.-A., Guo, Y.-J., and Su, Y.-S. (2023). Semi-automated behavior-driven testing for the web front-ends. In *2023 IEEE International Conference on e-Business Engineering (ICEBE)*, pages 225–230. IEEE.

Mock, M., Melegati, J., and Russo, B. (2024). Generative ai for test driven development: Preliminary results. *arXiv preprint arXiv:2405.10849*.

North, D. (2006). Introducing BDD. https://dannorth.net/introducing-bdd/.

North, D. et al. (2019). What's in a story? *Dosegljivo: https://dannorth. net/whats-in-a-story/[Dostopano 4. 5. 2016]*.

Oliveira, G., Marczak, S., and Moralles, C. (2019). How to evaluate bdd scenarios' quality? In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 481–490.

Pereira, L., Sharp, H., de Souza, C., Oliveira, G., Marczak, S., and Bastos, R. (2018). Behavior-Driven Development benefits and challenges: reports from an industrial study. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–4.

Rajbhoj, A., Somase, A., Kulkarni, P., and Kulkarni, V. (2024). Accelerating software development using generative ai: Chatgpt case study. In *Proceedings of the 17th Innovations in Software Engineering Conference*, pages 1–11.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering. Springer*, V.14:131–164.

Sauvola, J., Tarkoma, S., Klemettinen, M., Riekki, J., and Doermann, D. (2024). Future of software development with generative ai. *Automated Software Engineering*, 31(1):26.

Silva, T. R. and Fitzgerald, B. (2021). Empirical findings on BDD story parsing to support consistency assurance between requirements and artifacts. In *Evaluation and Assessment in Software Engineering*, pages 266–271.

Smart, J. (2014). *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, Shelter Island, NY.

Takerngsaksiri, W., Charakorn, R., Tantithamthavorn, C., and Li, Y.-F. (2024). Tdd without tears: Towards test case generation from requirements through deep reinforcement learning. *arXiv preprint arXiv:2401.07576*.

Wohlin, C., Höst, M., and Henningsson, K. (2003). Empirical research methods in software engineering. *Empirical methods and studies in software engineering: Experiences from ESERNET. Springer*, pages 7–23.

Zameni, T., van Den Bos, P., Tretmans, J., Foederer, J., and Rensink, A. (2023). From bdd scenarios to test case generation. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 36–44. IEEE.

Zhang, L., Wang, Y., and Li, X. (2023). Enhancing bdd test generation with large language models. *Journal of Software Engineering Research and Development*, 11(2):75–90.

Zhou, X., Jin, Y., Zhang, H., Li, S., and Huang, X. (2016). A map of threats to validity of Systematic Literature Reviews in Software Engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 153–160. IEEE.