Comparative Analysis of Large Language Model Tools for Automated Test Data Generation from BDD

Isela Mendoza
Universidade Federal Fluminense
Niterói, RJ, Brazil
imendoza@id.uff.br

Fernando Silva Filho
Universidade Federal Fluminense
Niterói, RJ, Brazil
fernandomsf@id.uff.br

Gustavo Medeiros
Universidade Federal Fluminense
Niterói, RJ, Brazil
gustavomedeiros@id.uff.br

Aline Paes
Universidade Federal Fluminense
Niterói, RJ, Brazil
alinepaes@ic.uff.br

Vânia O. Neves
Universidade Federal Fluminense
Niterói, RJ, Brazil
vania@ic.uff.br

ABSTRACT

Automating processes reduces human workload, particularly in software testing, where automation enhances quality and efficiency. Behavior-driven development (BDD) focuses on software behavior to define and validate required functionalities, using tools to translate functional requirements into automated tests. However, creating BDD scenarios and associated test data inputs is time-consuming and heavily reliant on a good input data set. Large Language Models (LLMs) such as Microsoft's Copilot, OpenAI's ChatGPT-3.5, ChatGPT-4, and Google's Gemini offer potential solutions by automating test data generation. This study evaluates these LLMs' ability to understand BDD scenarios and generate corresponding test data across five scenarios ranked by complexity. It assesses the LLMs' learning, assertiveness, response structuring, quality, representativeness, and coverage of the generated test data. The results indicate that ChatGPT-4 and Gemini stand out as the best tools that met our expectations, showing promise for advancing the automation of test data generation from BDD scenarios.

KEYWORDS

Test Automation, Large Language Models, Behavior-Driven Development, AI in Software Testing, Test Data Generation

1 INTRODUCTION

Software testing is one of the main activities for building quality software. Although it does not predict all errors that may occur during its development, it helps to find and avoid possible errors [12]. This activity verifies that the software under test produces the desired output based on a set of inputs and an execution environment. However, it is an expensive and time-consuming activity if performed manually; for this, both industry and academia have offered a wide range of tools that offer automated support for different types of software testing [17] [11] [13]. Among them, the Behaviour-Driven Development (BDD) [16] tools enable teams to write test cases in natural language, making them accessible to both technical and non-technical stakeholders [16].

This way, in test automation, BDD tools help convert functional requirements into automated tests, guiding analysts in verifying functionality and documenting system requirements in a more testable and understandable way for everyone involved. Since BDD helps translate functional requirements into automated tests, the efficiency of the development process is improved. Nonetheless, it is known that manually creating test scenarios traditionally requires considerable time and effort from test analysts [6]. According to experts in current companies, the corresponding set of input test data for BDD scenarios is also a significant time-consuming effort.

Furthermore, evaluating these requirements from BDD scenarios relies heavily on a suitable set of input test data that must be semantically coherent with the base structure and the restrictions to be used by automated software tests. Several works address the generation set of input test data - inputs to execute an algorithm or specific functionality, i.e., unit testing [5] [20][15]. However, to our knowledge, these works do not address the automatic generation of input test data, considering the semantics of the requirements to verify the system's behavior as a whole, such as system testing.

BDD allows functional specifications to be written in natural language that can be automatically transformed into executable test cases. Large Language Models (LLMs) [14] can play an important role in translating and interpreting requirements and test cases expressed in natural language, as is the case with BDD. LLMs like OpenAI's GPT (Generative Pre-trained Transformer) [1] and others of their kind, commonly called Chatbots, are trained on a vast amount of text to process and generate human language coherently and contextually relevantly. These tools can perform a wide range of tasks related to natural language processing, such as text generation and answering questions, among others. Their ability to accurately process and generate natural language allows them to capture complex relationships between words and phrases in a text.

In this context, the main objective of this work is to introduce an approach for automatically generating test data for BDD scenarios leveraging LLMs. Moreover, this work intends to evaluate the efficiency and effectiveness of LLM tools in understanding the context of these BDD scenarios and generating the respective set of input test data. This study aims to answer the following research questions (RQ):

RQ1: Can LLM tools generate sets of quality test data semantically coherent with the BDD scenarios' logic and context?

RQ2: Which tool most efficiently and effectively met our expectations?

To meet our objectives, we first propose an approach that establishes the general context of the research: the automated generation of test data for BDD scenarios using LLM tools. Next, we detail the

methodology, which describes how this approach will be implemented through an experimental study and comparative analysis between Microsoft's Copilot, OpenAI's ChatGPT-3.5, ChatGPT -4, and Google's Gemini, evaluating their effectiveness and efficiency in generating test data for BDD.

This approach would allow for faster adaptation to changes, as the flexibility of LLMs makes it easier to update test data as requirements and specifications change, keeping tests always up to date with minimal effort. Our proposal can help companies simplify, optimize, and improve their automatic testing processes. Automating test data generation significantly reduces test preparation time, allowing test analysts to focus on more critical tasks.

As a result of the experimental evaluation, ChatGPT-4 and Gemini were the tools that served us best, tying in performance and surpassing even testing experts in many aspects, followed by ChatGPT-3.5. These tools allowed us to generate data faster and with less effort than if it were created manually. Copilot did not meet our expectations, and we will not consider it in the approach due to its lower efficiency.

One of the main contributions of our study is the approach to automatically generate input test data from BDD scenarios using LLMs for optimizing time and effort without affecting data quality. The proposed methodology, which has a transparent and replicable structure, and the designed prompts are also considered significant contributions.

The paper is structured as follows: Section 2 introduces the concepts and terminology related to BDD and how the test automation process with BDD. Section 3 reviews related works in the field. Section 4 describes our approach proposed. Section 5 details the experiment methodology to validate our approach. Section 6 presents the results and discussion. Section 7 analyzes the threats to validity and limitations. Finally, Section 8 provides the conclusions, summarizing the main contributions and outlining potential future work.

2 BDD TERMS AND CONCEPTS

BDD (Behavior-Driven Development) is a technique that aims to integrate business rules with the programming language, focusing on software behavior [16]. It is used to describe the features needed to build software as well as test cases for automatic testing. This section will explain some concepts and terms related to BDD files and their structure and syntax.

Gherkin [16] is a domain-specific language designed to describe behaviors in a human-accessible way, grounded in a framework and essential keywords to ensure a clear and accurate description of system interactions.

The Gherkin language is widely adopted to describe software system behaviors due to its readability, structuring, standardization, ease of automation, and ability to serve as documentation. The main terms to specify how each step of interaction with the system is: **Feature:** Used to name the functionality to be tested in this feature. **Background:** When several scenarios exist in a feature film with the same precondition. **Scenario:** Describes the expected behavior (Then) given the preconditions (Given) and action (When) specified. **Given:** Used to specify a precondition before proceeding to the next steps. As a precondition, it is usually written in the past tense. **When:** Used when an action will be executed and a reaction is expected from the system, which will be validated in the "Then" step. This step is written in the present tense. **Then:** Validates whether the expected event happened. There is always a "When" step, as the reaction to the action received is validated here. Because it is the expected result, it is usually written in the form of the near future. **And:** If further interaction with the system is necessary to complement a flow, but it is not necessarily an action or reaction, "And" is used. **But:** Generally serves the same functionality as "And" but is usually used after a negative validation after "Then".

Other important keywords explicitly used for scenarios involving testing tasks are **Scenario Outline:** These are business scenarios with more than one "Example" used for automated testing. They will be executed on the same number of lines in the example table, each time using the information from the following line. In other words, the same scenario has the same flow but needs to use/validate different information. **Examples:** Always follow the "Scenario Outline", as the table with the data to be used to execute it is specified here. A "Scenario Outline" will run the same number of rows in the example table, each time using the data from the next row.

Scenarios written in Gherkin can be easily converted into automated tests using test automation tools, allowing tests to be run repeatedly and integrated into the ongoing development process. The next code shows an example of a scenario prepared for testing with the syntax and structure of the Gerking language:

```
Feature: Student grade
  Background: Teacher is logged into the system
  Scenario Outline: Verify the student average
    Given the student wasn't failed due to absence
    When the student has an average '<value>'
    And the average is less than 6
    Then the student is failed '<result>'
  Examples:
    | value | result |
    | " "   | Failure: Must be a natural number |
    | -7.5  | Failure: Must be a natural number |
    | 8.3   | Success: The student was passed |
    | 6.0   | Success: The student was passed |
    | 3.9   | Failure: The student was failed |
```

Figure 1 shows how BDD's automated testing process can occur. The test analyst receives functional documentation and system specifications through user stories in its initial stages. Based on this information, the analyst creates BDD files with the corresponding test scenarios to test the system's functionalities. BDD scenarios can require a set of input test data called "Examples"; for this, the next step consists of designing these data. Finally, once all BDD files have been designed and completed, the test suite is automatically created and executed with the test cases corresponding to these BDDs.

3 RELATED WORKS

In Software Engineering (SE), we see a growing interest in and increasingly widespread use of LLMs across multiple domains. Research conducted by [4] offers a comprehensive view of this trend, highlighting the diverse applications of LLMs in activities such

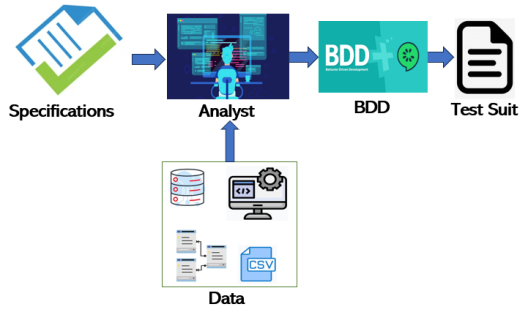

Figure 1: Manual data generation

as coding, design, requirements, repair, refactoring, performance improvement, documentation, and analysis within the SE. That survey also points to significant technical challenges associated with these advances, including reliable techniques to detect and correct incorrect solutions.

The rising use of LLMs in SE reflects the recognition of their broad potential in several areas, particularly software testing. Some studies have emerged exploring their application in this context, considering them a promising alternative worth exploring. For example, Wang et al. [18] provide a comprehensive review of the use of LLMs in the context of software testing. The study analyzes approaches that employ LLMs from both the software testing perspective and the models themselves, highlighting everyday tasks such as test case preparation, which includes unit test cases, test oracles, test inputs generated for the system test, and debugging and repairing programs.

Although several approaches already employ the use of LLMs in unit testing, few studies still perform functionality testing at the system level in relation to the generation of test inputs. Ye et al. [19] employed LLM to generate JavaScript programs, using the ECMAScript specifications to generate test data and test scripts automatically. Differential tests are then applied to expose bugs. Deng et al. [3] use LLM to extract crucial information related to the test scenario of a traffic rule. The extracted information is represented in a test scenario schematic. Subsequently, the corresponding scenario scripts are synthesized to construct the test scenario.

The authors in [18] point out that the generation of system-level test inputs for software testing varies depending on the type of software. For example, for mobile applications, test input generation requires providing a wide range of text inputs or combinations of operations, which is necessary to test the application's functionality and user interface. For mobile application testing, a crucial difficulty is generating suitable text inputs to advance to the next page, which remains a significant obstacle for test coverage. Traditional methods, like heuristic-based or constraint-based techniques, cannot understand the semantic information of the GUI page. In response, Liu et al. [7] employ LLMs to intelligently generate semantic input text, adapting it to the graphical user interface (GUI) context. LLM automatically extracts information from EditText-related components, generates prompts, and then uses them to generate text input. Alternatively, Liu et al. [8] approach test input generation for mobile GUI testing as a question-and-answer task involving the LLM in a conversation with the mobile applications. This process passes information from the GUI page to the LLM, which generates test scripts and runs them, continuing to provide feedback to the LLM and iterating the entire process. This approach extracts the static context of the GUI page and the dynamic context of the iterative testing process, allowing LLM to better understand the GUI page and the entire testing process.

Although using LLMs has become a significant topic in software testing research, previous studies have already explored the generation of test scripts within the scope of BDDs. Marques and Fernandes [9] adopted an approach to generating functional tests from scenarios built according to the BDD concept. They used Aspect-Oriented Programming (AOP) to map the internal system functions triggered during test execution, identifying parameters and feedback to generate relevant automated tests. This methodology resulted in high test coverage, improving system quality and facilitating developers' work by allowing the automatic execution of a wide range of tests. The results demonstrated that the approach not only reproduced the original tests but also generated additional tests considered relevant to the software requirements defined in the BDD.

All approaches discussed share common aspects, such as the need to generate input data semantically coherent with system specifications and constraints and their application in system testing. However, while some use LLM-based technologies to generate input test data, many are applied in specific projects and contexts with their specification file formats. Furthermore, studies that employ other methodologies to generate data generally do not consider the semantic relationship between data and system requirements and do not use BDD. On the other hand, works that use BDD are more focused on test cases in general, neglecting their specific test data. Moreover, they do not consider the textual description of the scenario when designing test data. In this sense, the approach presented in this article aims to fill this gap.

4 LLM-BASED APPROACH FOR AUTOMATIC GENERATION OF BDD EXAMPLES

To conduct the system testing process appropriately and in line with industry practices, it is essential to understand the semantic relationship between test input data and the scenarios, restrictions, and requirements to be tested.

Test input data originates from several sources, including relational databases and external files in diverse formats such as XML and CSV, tools, and other public knowledge bases, such as the Internet. When creating the input data set for the test, it is necessary to interact with several database tables with integrity checks. Additionally, the semantic connection between this data and the test requirements is usually available in functional documentation or BDD specifications. Therefore, the input test data must be semantically aligned with the base structure, constraints, and system logic. To solve this problem, we propose this approach by using LLM models capable of understanding the context of BDD scenarios and generating the respective set of input test data semantically coherent with the underlying logic of the tested functional requirement.

Designing the test input examples in test automation, currently in the industry, is carried out as shown in Figure 1. This process is usually time-consuming and prone to errors. Also, for this data to be used to test specific functionalities, it is subjected to a manual transformation process of selecting and reducing these data sets.

For selection, the data must be relevant: select specific data according to all the functionalities that will be tested. The data must also be of good quality, error-free, and complete. Finally, it must be valid: data must not be outdated or incoherent. Once the selection and data reduction are complete, obtaining a smaller, viable, and representative data set will be necessary. Their quality must be maintained, and their quantity must be reduced to cover all situations and combinations sufficient for the test cases. This process results in the set of test input data that will be used to constitute the test cases executed in automatic testing.

Our approach proposes to automate this process, replacing the manual work of the test analyst with an LLM tool. The automated testing process is illustrated in Figure 2. The test analyst receives functional documentation and system specifications through user stories in the early stages. Using this information, the analyst develops BDD files containing test scenarios to assess the system's functionalities. Each scenario in the BDD requires input test data, known as "Examples". The next step involves generating this data by incorporating LLM tools; for this purpose, it is essential to highlight that for these tools to return what we expect with the highest possible quality and accuracy, it is necessary to properly train them by employing prompt engineering, which we will explain in more detail in the following sections. Once all BDD files are finalized, the test suite is automatically generated and executed.

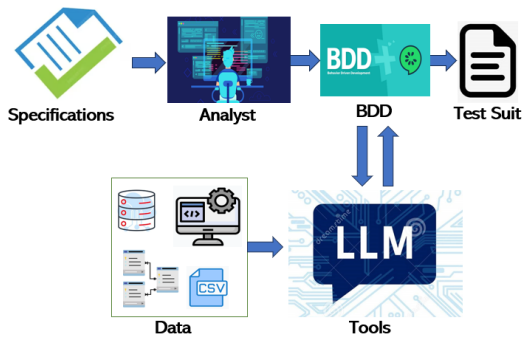

Figure 2: Data generation with LLM tools

This approach significantly relies on the ability of LLM tools to generate an appropriate dataset. With this in mind, we evaluated the major LLM tools available to determine their effectiveness in this context. The following sections describe the methodology and the results of this evaluation.

5 METHODOLOGY

This section outlines the methodology adopted to conduct our experiment. Through this methodology, we aim to validate our approach by evaluating the effectiveness and efficiency of various LLM tools to determine which best aligns with our expectations. Figure 3 illustrates the steps of our experiment. A brief description of each step is provided below, and more details are presented in the subsequent subsections.

Step-1 consists of selecting the LLM tools. We chose to analyze ChatGPT-3.5 and ChatGPT-4 from OpenAI, Copilot in Bing from Microsoft, and Gemini from Google, which have notable popularity and prestige as their development companies are strong competitors (Subsection 5.1). In Step-2, five BDD scenarios from different contexts are chosen and classified according to their complexity (Subsection 5.2). In Step-3, we have the benchmark with the input test data sets, which will be compared later with those generated by the tool (Subsection 5.3). Step-4 elaborates on the prompts to be executed in the LLM tools (Subsection 5.4). The execution of the prompts in the tools was carried out in April 2024. Finally, Step-5 comprises the evaluation of the responses obtained by the LLM tools (Subsection 5.5).

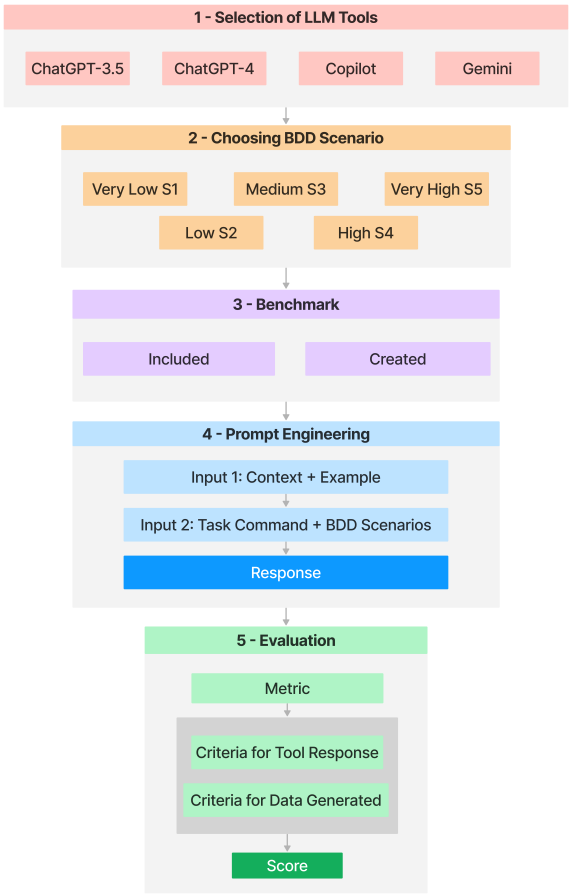

Figure 3: Workflow of the experiment methodology

5.1 Selection of LLM tools

The LLM tools chosen for this research include Microsoft's Copilot, OpenAI's ChatGPT-3.5, ChatGPT-4, and Google's Gemini. Although

there are other known tools with good performance in tasks and purposes similar to these, such as Llama, Vicuna, Claude, Mistral, and Alpaca, among others, we chose these tools due to their notable popularity and the prestige of the companies that developed them, which were recognized for their technical robustness. Products from these corporations are often considered safer, more reliable, and affordable. These factors and the ability to process and understand natural language on a large scale of these tools, representing a crucial milestone in Artificial Intelligence, guided our choice. Big Tech companies like Microsoft, OpenAI, and Google are at the forefront of this progress through their advanced tools and platforms.

Table 1 presents relevant characteristics of the LLM tools selected for the experiment, highlighting aspects such as the model used and the ability to perform web queries in real-time (internet consults), a feature that can always guarantee updated answers. Regarding access, most tools offer both a user interface and APIs to facilitate integration and use, except for Copilot, which currently does not have an official API; however, unofficial versions developed by third parties are available. Another crucial point is the availability of documentation, which is essential to fully exploit these platforms' resources.

Table 1: LLM tools feature

Tool	Model	Web Query	Access	Doc.
Copilot	Prometheus	Yes	UI	No
ChatGPT-3.5	GPT-3.5-turbo	Not	UI/API	Yes
ChatGPT-4	GPT-4-turbo	Yes	UI/API	Yes
Gemini	LaMDA	Not	UI/API	Yes

5.2 Choosing BDD scenarios

At this stage, a complexity scale is established to categorize the scenarios, ranging from very low to very high complexity, explained below, followed by their selection and classification. This categorization considers the scenarios' domain relevance and specificity, which can range from generic and common to highly specific. Additionally, the complexity of input data is considered, which can vary from simple data and integrated data from different sources to data requiring more complex mathematical reasoning for their generation.

Very Low complexity: It is defined based on widely recognized domains, making them easily understandable by the general public. It focuses on performing basic math operations and presenting simple input and output data.

Low complexity: At this level, more specific and specialized domains may require some degree of technical knowledge, although not necessarily specialization. Tasks at this level typically include manipulating variables and simple data types.

Medium complexity: This level of complexity may require integrating diverse, interconnected information. It involves more complex data types with interdependence that need to be understood.

High complexity: Complex tasks that demand expertise and skills in a specific area. Include mathematical calculations that require comprehension and application of knowledge to the context.

Very High complexity: Tasks at this level of complexity can be more challenging, involving logical reasoning in a specific mathematical domain that demands a more advanced understanding than the previous level and the types of data with specific nomenclatures.

As detailed below, five BDD scenarios (S1-S5) from various contexts were selected to align with this complexity scale. These scenarios include real cases from open-source repositories and company projects. Additionally, we created two fictitious scenarios (S4 and S5) designed to evaluate scenarios involving mathematical calculations and logical deduction in the High and Very High complexity categories. The objective of classifying and choosing this variety of scenarios is to evaluate the tools' effectiveness and efficiency in generating input test data.

Scenario 1 (S1): Very Low complexity

This scenario checks the validity of triangles by analyzing the lengths of their sides. Evaluates whether the given measurements can form a triangle based on geometric principles. Specifically, it checks whether the sum of the lengths of any two sides is greater than the length of the third side for each possible combination.

```
Scenario outline: Checking triangles
  Given that I have sides '<side_a>',
        '<side_b>' and '<side_c>'
  When  I check if it's a triangle
  Then  it should display '<result>'

Examples:
  |side_a|side_b|side_c|result|isValid|
```

Scenario 2 (S2): Low complexity

Focuses on scanning a target host for open TCP ports within a specific range. Verify that the configuration of the host and network is as expected.

```
Scenario outline: Host Configuration
  Given the target host name '<host>'
  When  TCP ports from '<startPort>'
        to '<endPort>' are scanned using
       '<threads>' threads and a timeout
        of '<timeout>' milliseconds
  And   the '<state>' ports are selected
  Then  the ports should be '<ports>'
Examples:
  |host|startPort|endPort|threads
  |timeout|state|ports|isValid|
```

Scenario 3 (S3): Medium complexity

The functionality describes registering a new location in a system that needs to deal with interconnected geographic location information, making it essential to generate accurate, correct, authentic, and valid data.

```
Scenario outline: Register location
  Given that the user is logged in
        as an administrator
  And   accesses the homepage
  And   click on the 'Locations' button
  And   click on the 'New Location' button
  And   fill in '<Description>', '<Zip Code>',
```

```
        '<Address>', '<Number>', '<Complex>',
        '<Neighborhood>', '<City>', '<State>',
        '<Country>', '<Latitude>' and '<Longitude>'
  When  the 'Save' button is clicked
  Then  the user is redirected to the
        'List of Locations' page
  And   the last registered location has
        the name '<Description>'
  And   delete the last location of
        description '<Description>'
Examples:
  |Description|ZipCode|Address|Number
  |Complement|Neighborhood|City|State
  |Country|Latitude|Longitude|isValid|
```

Scenario 4 (S4): High complexity

Describes the nature of a problem that involves logical reasoning in a mathematical context. The math problem is about a paint-mixing machine that creates shades. Check if the amount of colors to be increased results in the correct shade.

```
Scenario outline: Check the amount of paint
  Given a machine that mixes paints of different
        colors to obtain different shades
  When  someone buys <total_liters> liters of
        paint of one shade from a mixture of
        '<liters_colorX>' liters of color X and
        '<liters_colorwhite>' liters of white color
  And   the quantity of paint purchased needed
        to be increased, buying more '<value>'
        liters of the same paint mixture with
        the same shade is necessary.
  Then  the amount of paint in color X to preserve
        the same tones when mixed with the
        color white is the '<result>' liters
Examples:
  |total_liters|liters_colorX
  |liters_colorwhite|value|result|
```

Scenario 5 (S5): Very High complexity

This scenario is more complex linear algebra calculations, the inner product of two degree-2 polynomials. It provides the polynomials and expects the solution based on their values at specific points (-1, 0, and 1).

```
Scenario outline: Inner product
  Given two polynomials of degree 2,
        '<p1>' and '<p2>' the inner product
        given by calculating the value of
        the polynomials at -1, 0 and 1
  When p(t) = '<p1>' and q(t) = '<p2>'
  Then solve <p,q> by calculating the value
       of the polynomials at -1, 0 and 1
Examples:
  | p1 | p2 | result |
```

5.3 Benchmark

The benchmark consists of two types of data: "Included" data, which are reference data extracted from real industry projects, and "Created" data, developed by the researchers of this study for scenarios that do not have example data. Scenarios S2 and S3 were included from real industry projects. In these cases, the test analysts responsible for creating these test data provided the "Examples" for each scenario.

Scenarios S1, S4, and S5 were created by our research team through desk checking. This manual process is performed by testing specialists to validate the logic of a specific functionality, resulting in a manually created table with the input and output variable values of the program. To do this, we manually created this table of values for each of the variables in each scenario, following the Gherkin language and the structure of the "Example" step, and applying black-box testing criteria such as Boundary Value Analysis (BVA) and Equivalence Partitioning (EP), to assign different data combinations to these variables. Both sets of data, obtained manually from the real and fictitious project scenarios, are later compared with those generated by the tools.

5.4 Prompt engineering

Prompt engineering involves providing specific commands to the LLM tool to direct the model on how to answer a question, influencing its behavior toward desired results without changing the model's weights. We developed a specific prompt for this experiment using the zero-shot-learning and few-shot-learning prompt techniques, following the corresponding guidelines in [2]. According to Wang et al. [18], zero-shot-learning and few-shot-learning techniques are the most prevalent in software testing applications.

There are two main types of zero-shot-learning [18]. The first, "No instructions", provides the model with just the task text to obtain the results directly. The second, "With instructions", involves giving the template specific headers or guidelines about the expected result. For example, a scenario is provided, and the tool is asked to generate "Examples" suitable for that scenario. Few-shot-learning presents a set of explanations and demonstrations with inputs and desired results about the target task; for example, complete scenarios are provided, presenting their structure and syntax with their valid and invalid test data from the "Examples" table. As the model sees the examples for the first time, it can better understand human intent and the criteria for the desired response types. It is essential for tasks that are more complex and intuitive for LLM that involve specific contexts like this.

In this experiment, the prompts were developed and refined until the final version was reached. Initially, we provided only the scenario and requested the generation of test data, which resulted in repeated and limited combinations. Therefore, we incorporated black-box testing criteria in the subsequent prompts. Additionally, since the results were not presented in standardized formats, we specified the structure of the BDD language we needed. Finally, we included specific examples to ensure that the final prompt was robust, meeting our expectations, and producing data with the desired accuracy and quality. The process to achieve the final prompt

is detailed in the flowchart in Figure 4, which shows the recommended sequence of instructions for generating the input test data for BDD scenarios.

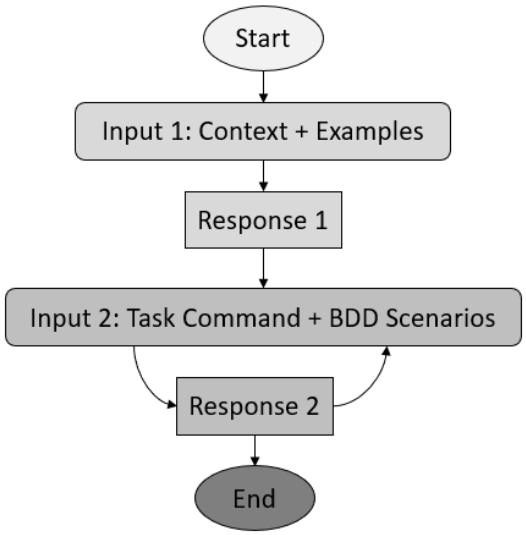

Figure 4: Prompts flow in the data generation process.

The proposed prompt is composed of two main elements. First, Input1 prepares the tool by providing context, explanations, and examples of BDD scenarios, including their language and structure. Interaction with each tool begins by entering Input1 and waiting for confirmation of understanding, usually an 'OK'. Second, after confirmation, we proceed with Input2, which addresses specific commands related to the context of each functionality, the order of tasks, and the corresponding BDD scenario. For each scenario, Input2 is repeated. The prompts used are:

Input 1: Context and Examples:

> *In the following prompts, receive the Scenarios corresponding to a given functionality as input; based on this information, only the set of test data pertinent to the 'Examples' table of each scenario as an output will be generated, with a set of values to meet the criteria of 'Equivalence Partitioning' and 'Boundary Value Analysis'.*

> *Create the 'Examples' table below each 'Scenario' that will be provided, where each column header corresponds to the variables defined in the 'Scenario' represented between smaller than (<) and greater than (>) signs. Make sure the answer correctly follows the syntax and structure of the Gherkin language.*

> *Do not chat with the user or make comments; return just the request. Below are examples to better understand the structure of a BDD 'Scenario' that includes 'Examples' data.*
>
> **(Here two examples of BDD scenarios are presented)**
>
> *Reply OK if it was clear and understandable.*
>
> TOOL RESPONSE: **'OK'**

Input 2: Task command and BDD Scenarios:

> *Knowing that a triangle can be equilateral, isosceles, scalene, or NULL.* ***Complete the example below with valid and invalid cases.***
> ***(Followed by the S1 script)***
> TOOL RESPONSE: **(Example table for this scenario)**

> *Knowing that startPort, endPort, threads, and timeout are integers, host an IP/site, the State is open or closed, and the ports are the ports.* ***Complete the example below with valid and invalid cases.***
> ***(Followed by the S2 script)***
> TOOL RESPONSE: **(Example table for this scenario)**

> *The description is the name of the Address; the ZIP code is the postal address code; the Address is the street/location; the number is an integer; the Complement is NULL or some letter; Names of neighborhood, city, and country; State is the abbreviation of the State, and Latitude and Longitude are geographic coordinates.* ***Complete the example below with valid and invalid cases.***
> ***(Followed by the S3 script)***
> TOOL RESPONSE: **(Example table for this scenario)**

For scenarios S4 and S5, it was enough to define the task order and the BDD scenario without additional specifications. The full prompt and results are available in our repository [10].

5.5 Evaluation

The results are evaluated based on the accuracy of each tool's response, ensuring that the data correctly reflects the information or scenarios intended for the test compared to the defined benchmark (desk check), which is also evaluated. In this section, we establish the metrics and the criteria based on response prediction, quality, diversity of test data, and coverage of test cases described in the BDD to evaluate the results and compare them with the benchmark. The criteria are organized into two groups: i) rating criteria for tool responses and ii) rating criteria for the set of input test data generated. Furthermore, we describe the procedure for analyzing these data.

The criteria use a Likert satisfaction scale as a metric. The definition of this metric and the evaluation of the criteria were conducted by the authors of this article, all of whom were actively involved in the study. Among them are two early career researchers and

one doctoral candidate in software testing, supervised by senior researchers, who are also authors of this work.

5.5.1 Metrics. The metric used to evaluate the criteria is based on the Likert scale.

The values that define the scale are: 0-Dissatisfied; 1-Minimally satisfied; 2-Partially satisfied; 3-Mostly satisfied; and 4-Completely satisfied. Each tool will be evaluated for each established criterion, determining to what extent the tool's response meets that criterion.

5.5.2 Criteria for tool response (TR). The criteria defined to evaluate tools responses are Learning (L), Assertiveness(A), and Structure (S), as explained below:

Learning (L): This criterion evaluates the tool's ability to assimilate and understand information from the examples and contexts provided. Measuring how the tool processes and integrates new data to produce responses that align with expectations is essential. Effectiveness in this criterion indicates that the tool can adapt and learn continuously, improving its accuracy and relevance in subsequent interactions.

Assertiveness (A): This criterion measures the relevance and adequacy of the tool's answers. An assertive response must not only directly address the question being questioned but also respect the context in which the question was asked, implying that the tool must discern nuances of context and provide technically correct and contextually appropriate responses. Assessing assertiveness is crucial to ensure the tool is valuable and effective in practical situations, avoiding generic or decontextualized responses.

Structure (S): This criterion evaluates the format of the responses generated by the tool, checking whether they follow an appropriate structure based on the example information previously provided regarding a BDD scenario and its different stages, mainly in the "Examples" stage, which contains the set of input test data, respecting the Gherkin language. This criterion ensures that the tool not only correctly absorbs the required format but also applies this knowledge in a practical and structured way, facilitating integration and understanding. Evaluating the framework in this context is crucial to ensure that responses are helpful and targeted in BDD test environments.

5.5.3 Criteria for the test data generated (DG). To evaluate the test data generated, we considered the following criteria:

Quality (Q): This criterion evaluates the integrity of the set of input test data generated by the tools. Data must be correct, consistent, valid, complete, and relevant to the test. Quality is crucial to ensure that test data can be effectively used to evaluate system functionality without errors that could compromise the integrity of test results.

Representativeness (R): Refers to the ability of the set of input test data to accurately reflect the behavior and characteristics of the functionality being tested. The set of input test data must be representative to be applied to the test environment and ensure accurate and valid test results. Lack of representation can lead to misleading results, negatively influencing system development or maintenance decisions.

Coverage (C): It involves the extent to which the set of input test data covers the features. Good coverage means the test adequately explores the potential variations or states the scenario intends to test. Coverage is essential to ensure that no significant aspects of the system are overlooked. A comprehensive set of input test data helps identify hidden vulnerabilities in less apparent areas and ensures the system is thoroughly tested.

By generating a set of input test data ensuring compliance with these criteria, we can ensure that test results are reliable, comprehensive, and valuable. Each criterion is essential to validate the quality and effectiveness of the tests.

5.5.4 Data Analysis Procedures. The results were analyzed both quantitatively and qualitatively, as explained below. All data are available in our GitHub repository [10].

Quantitative analysis: We assigned a score to each tool's response, indicating how well the response met each criterion. Then, we added the values assigned to each criterion and considered the tool with the highest total value as the one that best met our expectations.

The initial results were peer-reviewed to reach a consensus on the score to be assigned. In cases where there was no consensus, the senior researcher in the testing area reviewed and analyzed these cases, discussing them with everyone. After achieving an adequate level of agreement on the initial results, the task was divided, and the remaining results were evaluated individually.

The number of set test data generated for each scenario per tool and the benchmark are counted; then, the data coverage percentage is calculated using formula (1), where, for each scenario, the number of 'Unique set of test data' is divided by the 'Total set of test data' generated.

$$\textbf{\textit{Coverage}} = \frac{\textit{Unique set of test data}}{\textit{Total set of test data}} x\ 100 \quad (1)$$

Qualitative analysis: The responses generated by the tools for each scenario were analyzed and discussed based on the criteria defined and followed by a thorough analysis of the responses according to the complexity of the scenarios and in comparison with the benchmark.

6 RESULTS AND DISCUSSION

The scores awarded to each LLM tool and the benchmark for each criterion are shown in Table 2. The LLM that achieved the highest score was ChatGPT-4 and Gemini, demonstrating the best performance and satisfaction in all criteria, followed by ChatGPT-3.5. Copilot came in last place with the lowest score.

Table 2: Evaluation scores by criterion

Criteria type	TR	TR	TR	DG	DG	DG	TOTAL
Tools/Criteria	L	A	S	Q	R	C	SCORE
Copilot	3	4	2	4	2	2	17
ChatGPT-3.5	4	4	4	4	3	3	22
ChatGPT-4	4	4	4	4	4	3	23
Gemini	4	4	4	4	3	4	23
Benchmark	4	4	4	4	3	3	22

The graph in Figure 5 illustrates the number of set of test data generated for each scenario per tool and the benchmark. The graph

in Figure 6 reveals the percentage of the input data sets generated by the tools concerning coverage.

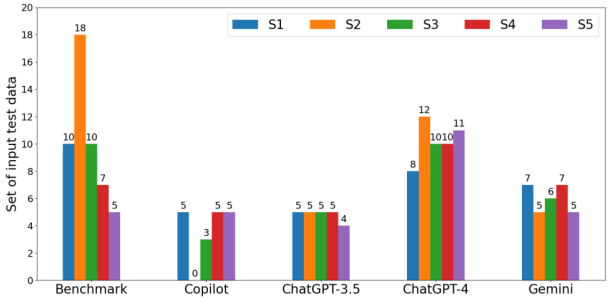

Figure 5: Number of set input test data

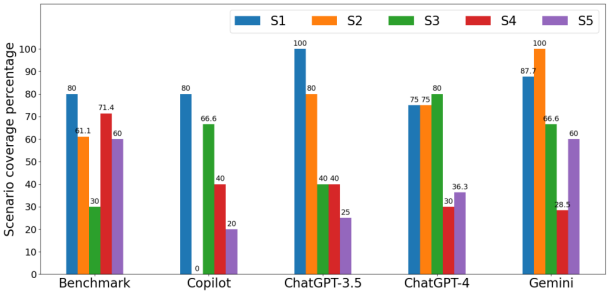

Figure 6: Coverage of each scenario for tools

6.1 Analysis based on evaluation criteria

According to the **Learning (L)** criterion, all tools adequately understood the context and examples provided, responding as expected, except Copilot. The latter only partially interpreted the instructions, misinterpreting the context since the first three scenarios (S1, S2 e S3) returned results in a table format as shown in Table 3, corresponding to scenario S1 of Copilot, unlike the requested BDD format, as specified in the prompt. However, in the last two scenarios, Copilot delivered responses adequately. This deviation from the initial format negatively impacted the evaluation of the **Structure (S)** criterion, which checks whether the format respects the BDD Gherkin language. Regarding **Assertiveness (A)**, all the tools performed well, successfully discerning the nuances of each scenario context and providing appropriate responses concerning context.

Up to this point, the criteria (L, S, A) have been analyzed, focusing mainly on the content of the answers provided and whether they are appropriate to the question. The following criteria analysis focuses on evaluating responses from the perspective of the input test data, checking whether it is sufficient to test the scenario in question.

About the **Quality (Q)** criterion of the set of test data, all tested tools generated correct, no errors, valid, and reliable data aligned with expectations regarding data quality, regardless of quantity. Regarding **Representativeness (R)**, which measures how much

Table 3: Incorrect structure response Copilot scenario 1

Table side_a	side_b	side_c	result	isValid
3	4	5	Success: It's a valid	true
2	2	5	Failure: It's not a valid	false
6	6	6	Success: It's a valid	true
1	2	3	Failure: It's not a valid	false
0	0	0	Failure: It's not a valid	false

the generated data reflects the behavior of the functionality described in the BDD scenarios, ChatGPT-4 obtained better efficiency in returning accurate data. In fulfilling this criterion, Copilot was the worst; in the low complexity scenario S2, no data was returned, consequently affecting the coverage criterion.

Finally, most tools satisfied the **Coverage (C)** criterion, adequately exploring critical parts of the functionality described in the scenarios. Gemini demonstrated superior performance compared to other tools in terms of coverage. The BVA and EP criteria employed by Gemini enabled the generation of a comprehensive range of data combinations and variations, leading to enhanced coverage. Note that the benchmark produced more test data sets in quantity, as shown in the graphic of Figure 5; this does not imply that they have the best sets. Coverage, represented in the graph of Figure 6, reveals the percentage of the best sets of input data generated by the tools about coverage per scenario.

6.2 Analysis considering the scenarios' complexity

Analyzing the tools' responses according to the complexity of the tested scenarios, we found that ChatGPT-3.5 and Gemini met our expectations, providing appropriate responses aligned with the context and equivalent to the benchmark. On the other hand, Copilot showed deficiencies in the first three scenarios, as explained above; however, its effectiveness improved in the last two scenarios, S4 and S5, with high and very high complexity, respectively, and remaining within the benchmark.

The third scenario S3, of medium complexity, presented a unique challenge. In this case, the benchmarks provided valid dummy data: the geographic coordinates. Interestingly, all the tools we tested outperformed the benchmarks in this scenario. Each provided several valid real-world examples, demonstrating their ability to go beyond a programmed response and generate creative and relevant results, underlining the capacity of these large linguistic models to replicate information, analyze it, and interpret it in novel ways.

The tool that stood out in all scenarios was ChatGPT-4, demonstrating greater efficiency in providing accurate responses, especially in the S3, where it took advantage of its ability to consult online information and browse the web to provide accurate and authentic data, saving time and human effort. Additionally, ChatGPT-4 outperformed the benchmark and the other tools in the latest very high complexity S5 scenario involving mathematical logic, showing significant potential in providing a substantial variety of input test data sets and accurate solutions, excelling in tasks that would require solid knowledge of linear algebra and more time and effort

if done manually. The following code presents part of ChatGPT-4's response to the S5 scenario, which includes complex polynomial operations and calculations performed efficiently, surpassing the human ability to produce similar data with the same efficiency.

Examples:

```
| p1   | p2   | result |
|"t^2"|"t^2"| 3      | # (1^2 + 0^2 + 1^2) = 3
| " "  |"t^2"|error| # Invalid polynomial p1
|"t^2"| " "  |error| # Invalid polynomial p2
```

7 THREATS TO VALIDITY AND LIMITATIONS

Some threats to validity and limitations could be associated with subjective bias when evaluating the scenarios. However, this was overcome, given that several authors evaluated the results. The diversity and experience of evaluators are crucial to mitigating this issue. However, even so, they do not eliminate the risk of subjectivity.

Another limitation is associated with the exhaustiveness of the generated scenarios, which could be the possibility that the scenarios generated by LLMs are not completely extensive. Language models may not cover all possible variables and situations within a test scenario. Human oversight is still required to ensure the generated scenarios are comprehensive and adequately represent the complexities and nuances of the domain.

Models adjusted to the specific context. Models that have yet to be trained with domain-specific data or examples may not produce results that are as accurate or relevant to the particular requirements of BDD scenarios. Therefore, the model must be adjusted or customized appropriately for the specific context of use to ensure better accuracy and relevance of the generated examples.

Also, threats to validity may arise due to LLMs' volatility and continuous evolution. Frequent updates to these models can result in variations in the results of repeated experiments over time, even when performed under identical conditions. Stability in model versions, accessibility, and adequate documentation are essential to ensure the reproducibility and replication of experiments.

8 CONCLUSION

This study proposed integrating BDD with LLM tools to generate semantically coherent test data in BDD scenarios. Additionally, a comparative evaluation was conducted between tools that could be used in this approach. A prompt was defined to be executed in these tools to conduct this evaluation, and criteria and metrics were established to assess the results.

In response to **RQ1:** *Can LLM tools generate sets of quality test data semantically coherent with the BDD scenarios' logic and context?*, we discovered that these tools greatly reduce the time and effort required by evaluators by delivering quality, representative data. This makes them a compelling alternative for companies to explore. The responses generated by these tools were equivalent to, and in many cases surpassed, those of a testing expert. Furthermore, the time and effort to obtain this data through the tools was considerably less and faster than creating it manually, given the difficulty in obtaining various valid, correct, and viable data for testing.

In response to **RQ2:** *Which tool most efficiently and effectively met our expectations?*, we found that the most efficient and practical tools that met our expectations were ChatGPT-4 and Gemini. Although ChatGPT-4 is a paid tool and provides very good resources, Gemini stood out as the best free option, scoring equally high. ChatGPT-3.5 also proved a viable alternative, with performance close to the others. Among the tools analyzed, only Copilot failed to produce satisfactory results. Consequently, due to its lower performance compared to the other tools, Copilot will not be considered in our approach.

Among the main contributions of our study, we highlight, firstly, the proposed approach for the automatic generation of input test data from BDD scenarios, which are semantically linked to the logic of functional requirements for system testing. Large Scale Language Models (LLM) tools significantly optimize the time and effort of companies' test analysts without compromising test data quality, which in many cases proved to be more representative than that created manually. Furthermore, the study offers a well-structured and developed methodology, facilitating its replication in future studies. The prompt designed and the techniques used for this, which proved to be the most appropriate and best met our expectations, is also considered one of our significant contributions.

For future work, we propose continuing the evaluation of other available tools. Additionally, an immediate step is to develop a plugin that can be integrated into Integrated Development Environments (IDEs) to facilitate the examples generation process for testers. We also intend to test the approach's effectiveness in real-world industry scenarios to validate its applicability and efficiency in production environments. It is essential also to consider integrating our approach with the existing software testing approaches using LLMs. According to the classification of the approaches presented in [18], our proposal is aligned with the system testing level and with the "Test Case Preparation" and "Test Execution" stages of the life cycle of software testing. This alignment makes integrating our approach with existing methods at different levels and stages of the software testing process more accessible. Finally, comparing the time and effort of manually generating test data versus the preparation, execution, and human supervision of the prompt could prove compelling.

ACKNOWLEDGMENTS

This paper has been supported by CNPq - *National Council for Scientific and Technological Development* (grants 143289/2021-7, 420025/2023-5, and 307088/2023-5) and FAPERJ - *Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro* (processes SEI-260003/000614/2023 and SEI-260003/002930/2024).

REFERENCES

[1] Valentina Alto. 2023. *Modern Generative AI with ChatGPT and OpenAI Models: Leverage the capabilities of OpenAI's LLM for productivity and innovation with GPT3 and GPT4.* Packt Publishing Ltd.

[2] DAIR.AI. 2024. Prompt Engineering Guide. https://www.promptingguide.ai/techniques.

[3] Yao Deng, Jiaohong Yao, Zhi Tu, Xi Zheng, Mengshi Zhang, and Tianyi Zhang. 2023. Target: Automated scenario generation from traffic rules for testing autonomous vehicles. *arXiv preprint arXiv:2305.06018* (2023).

[4] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE).* 31–53.

[5] Roger Ferguson and Bogdan Korel. 1996. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*

(TOSEM) 5, 1 (01 1996), 63–86.

[6] Dorothy Graham and Mark Fewster. 2012. *Experiences of Test Automation: Case Studies of Software Test Automation.* Addison-Wesley.

[7] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367.

[8] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. *arXiv preprint arXiv:2305.09434* (2023).

[9] Nicholas Nishimoto Marques and Rafael Alves Fernandes. 2020. Um arcabouço para a geração automatizada de testes funcionais a partir de cenários BDD. Bachelor in Computer Science.

[10] Isela Mendoza, Fernando Silva Filho, Gustavo Medeiros, Aline Paes, and Vânia O. Neves. 2024. Data Repository for Comparative Analysis of LLM Tools in BDD Test Data Generation. https://github.com/bbd-test-research/LLM-Tools-BDD-Test-Data Companion repository.

[11] Jean Carlos P. Miranda, Hugo T. Almeida, and Vânia O. Neves. 2018. PySoCA - Python Source-code Coverage and Analysis. In *Anais do IX Congresso Brasileiro de Software (CBSoft 2018) - Sessão de Ferramentas*. São Carlos/SP.

[12] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing.* John Wiley & Sons.

[13] Vânia O. Neves, Marcio E. Delamaro, and Paulo C. Masiero. 2017. Pateca: uma ferramenta de apoio ao teste estrutural de veículos autônomos. In *Anais do VIII Congresso Brasileiro de Software (CBSoft 2017) - Sessão de Ferramentas*. SBC, Porto Alegre, BR, 57–64.

[14] Mohaimenul Azam Khan Raiaan, Md Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. 2024. A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. *IEEE Access* 12 (2024), 26839–26874.

[15] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[16] John Ferguson Smart and Jan Molak. 2023. *BDD in Action: Behavior-driven development for the whole software lifecycle.* Simon and Schuster.

[17] Auri Marcelo Rizzo Vincenzi, Márcio Eduardo Delamaro, Arilo Claudio Dias Neto, Sandra Camargo Pinto Ferraz Fabbri, Mário Jino, and José Carlos Maldonado. 2018. *Automatização de teste de software com ferramentas de software livre.* Elsevier Brasil.

[18] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[19] Guozhu Ye, Zhiqiang Tang, Shih-Hao Tan, Shiqi Huang, Dongdong Fang, Xiaoyang Sun, Lei Bian, Haibo Wang, and Zhendong Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 435–450.

[20] Ruilian Zhao and Qing Li. 2007. Automatic test generation for dynamic data structures. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*. IEEE, 545–549.