Agentic AI for Behavior-Driven Development Testing Using Large Language Models

Ciprian Paduraru[1], Miruna Zavelca[1] and Alin Stefanescu[1,2]
[1]*Department of Computer Science, University of Bucharest, Romania*
[2]*Institute for Logic and Data Science, Romania*
{*ciprian.paduraru, miruna-andreea.zavelca, alin.stefanescu*}*@unibuc.ro*

Keywords: Behavior-Driven Development (BDD), Large Language Models (LLMs), Test Automation, Natural Language Processing (NLP), Software Testing Frameworks, Human-in-the-Loop, Agentic AI.

Abstract: Behavior-driven development (BDD) testing significantly improves communication and collaboration between developers, testers and business stakeholders, and ensures that software functionality meets business requirements. However, the benefits of BDD are often overshadowed by the complexity of writing test cases, making it difficult for non-technical stakeholders. To address this challenge, we propose BDDTestAIGen, a framework that uses Large Language Models (LLMs), Natural Language Processing (NLP) techniques, human-in-the-loop and Agentic AI methods to automate BDD test creation. This approach aims to reduce manual effort and effectively involve all project stakeholders. By fine-tuning an open-source LLM, we improve domain-specific customization, data privacy and cost efficiency. Our research shows that small models provide a balance between computational efficiency and ease of use. Contributions include the innovative integration of NLP and LLMs into BDD test automation, an adaptable open-source framework, evaluation against industry-relevant scenarios, and a discussion of the limitations, challenges and future directions in this area.

1 INTRODUCTION

Motivation. Behavior Driven Development (BDD) testing is an important practice because it promotes clear communication and collaboration between developers, testers and business stakeholders and ensures that software functionality closely aligns with business requirements and user needs. With this method, human-readable test case descriptions can be written and linked to existing code source databases.

In our research, we found that while BDD aims to use natural language, the scenarios can still be complex and difficult for non-technical stakeholders to understand. Non-technical stakeholders may need to be trained to write and interpret BDD tests effectively, which can be time and resource consuming. (Smart and Molak, 2023). This also applies to programmers who may struggle to combine the possibilities of what is testable in a large codebase with appropriate interaction with the test framework (e.g. steps vs. function calls in code in Gherkin syntax). With this in mind, we worked with the local game industry of Amber[1] and Gameloft[2] to understand the issues involved in the adoption of BDD testing procedures by all stakeholders involved in project development. This is particularly motivated by the fact that in the games industry, the typical ratio of non-programmers (quality assurance - QA, artists, designers, audio specialists) to programmers is about 4:1 (Shrestha et al., 2025).

Goals and Innovations. Our vision is that Large Language Models (LLMs) can significantly improve BDD testing practice by automating their creation. This approach could reduce the time and effort required for manual test creation and improve scalability by making it easier to involve all stakeholders of a project in the process.

With this in mind, we propose *BDDTestAIGen*, a framework that uses LLMs, basic natural language processing (NLP) techniques, and human-in-the-loop to establish the link between user specification, steps, and source code implementation in a software product. According to our research, this is the first work that combines the above techniques to support BDD test generation. The methods used are novel as we attempt to combine the power of LLMs in reason-

[1]https://amberstudio.com

[2]https://gameloft.com

Paduraru, C., Zavelca, M. and Stefanescu, A.
Agentic AI for Behavior-Driven Development Testing Using Large Language Models.
DOI: 10.5220/0013374400003890

In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 2*, pages 805-815
ISBN: 978-989-758-737-5; ISSN: 2184-433X

ing and generation with agents, internal and external tools (Yao et al., 2022), and source code reflection. First, we justify our contribution to fine-tuning an open-source LLM within the core of the framework and its advantages compared to relying solely on external models such as GPT-4 (OpenAI et al., 2024). Fine-tuning an internal LLM allows organizations to adapt the models to their needs and improve consistency with domain-specific knowledge and terminology (Lv et al., 2024). This approach improves data protection, reduces long-term costs, and provides more control and flexibility when updating and deploying models. The methods also consider the practicality of using the tool in production. The goal is therefore to develop a small open-source model that can run on both the CPUs and GPUs of end users. In this sense, during our experiments, we found that a class 7B/8B model (eight billion parameters) such as Llama3.1 (Grattafiori et al., 2024) can provide the trade-off between computational efficiency and ease of use.

Working Mode. The interaction between users and the framework takes place via a conversational *assistant* that works in the following way. The complete test creation is coordinated step by step, with the human giving instructions in natural language. The assistant then helps with the correct syntax and considerations as to which available implementation in the source code of the internal project the natural language instruction can refer to, using LLM.

The tool avoids the problems observed in the previous work (Karpurapu et al., 2024), where the most common syntax errors concerned the absence of certain keywords, the wrong order of parameters, names or even a wrong format. The proposed LLM can draw conclusions based on the features processed with some core NLP techniques and fix these problems in almost all aspects. In our context, the problem of missing links (i.e. incorrectly linked step implementation) is solved using human-in-the-loop, as the assistant asks for help by informing the user that it could not find a link to the implementation. In addition, the user can edit the generated response if the LLM suggestions are incorrect.

In the following we summarize the contribution of our work:

- According to our research, this is the first study to investigate the use of LLMs in combination with NLP, human-in-the-loop, and Agentic AI techniques to automate BDD test creation in an industrial setting, reducing manual effort and engaging all project stakeholders more effectively. The concept of Agentic AI (Kapoor et al., 2024) is used to allow the LLM reason and combine tools (function) calling iteratively to solve the requests. By using the human-in-the-loop concept, users have full control over the generation, with the AI assistant acting as a helper in this process.
- The proposed methods and evaluation examples were developed following discussions with the industry to understand the gaps in improving product testing. In our case, we used two public games in the market. According to our observations, the BDD tests can be written with AI autocorrection, similar to how LLMs help in software development through code autocompletion.
- The proposed framework called *BDDTestAIGen* is available as open source at https://github.com/unibuc-cs/BDDTestingWithLLMs.git. It has a plugin architecture with scripts to adapt to new use cases, and domains or to change components (e.g. the LLM model) as required. A Docker image is also provided for faster evaluation.
- We consider the computational effort that is justified to use the methods on developer machines. From this perspective, we evaluate and conclude that small models (such as Llama3.1 8B), fine-tuned and combined with various NLP feature processing and pruning techniques, can provide the right balance between cost and performance.

The rest of the article is organized as follows. The next section presents related work in the field and our connection or innovations to it. A contextual introduction to BDD and the connection to our goals and methods can be found in Section 3. The architecture and details of our implementation can be found in Section 4. The evaluation in an industrial prototype environment is shown in 5. The final section discusses the conclusions.

2 RELATED WORK

Before the LLM trend, a combination of AI and specific NLP techniques was used to improve BDD test generation. The review paper in (Garousi et al., 2020) discusses how common NLP methods, code information extraction, and probabilistic matching were used to automatically generate executable software tests from structured English scenario descriptions. A concrete application of these methods is the work in (Storer and Bob, 2019). The methods were very inspiring, as their use together with LLMs can, in our experience, increase computational performance.

A parallel but related and insightful work is (Ouédraogo et al., 2024), in which the authors in-

vestigate the effectiveness of Large Language Models (LLMs) in the creation of unit tests. The study evaluates four different LLMs and five prompt engineering methods, with a total of 216,300 tests performed for 690 Java classes. The results show the promise of LLMs in automating test creation, but also highlight the need for improvements in test quality, especially in minimizing common test problems. We address these recommendations through human-in-the-loop.

In the area of test case generation, the authors in (Li and Yuan, 2024) explore the idea of using Large Language Models (LLMs) for general white-box test generation and point out their challenges in complex tasks. To address these issues, the authors propose a multi-agent system called TestChain that improves the accuracy of test cases through a ReAct (Yao et al., 2022) format that allows LLMs to interact with a Python interpreter. We adopt their technique of using ReAcT and the live Python interpreter and test execution, but improve their methods by fine-tuning a model specifically for BDD cases and incorporating expert knowledge (humans) into the test generation loop.

The study in (Karpurapu et al., 2024) investigates the use of large language models (LLMs) to automate the generation of acceptance tests in BDD. Using null and few prompts, the study evaluates LLMs such as GPT-3.5, GPT-4, Llama-2-13B, and PaLM-2 and finds that GPT-4 performs excellently in generating error-free BDD acceptance tests. The study highlights the effectiveness of few-shot prompts in improving accuracy through context-internal learning and provides a comparative analysis of LLMs, highlighting their potential to improve collaborative BDD practices and paving the way for future research in automated BDD acceptance test generation. The motivation for our work stems from and continues this research by fine-tuning an open-source model, combining it with previous techniques from the literature, and finally giving the human stakeholder full control over the generation process.

3 BDD METHODOLOGY AND ITS APPLICATION

Behavior-driven development (BDD) (Irshad et al., 2021) is a methodology in software development that aims to foster collaboration between different stakeholders, including developers, QA teams, and business or non-technical participants. It is often used as part of agile development methods. BDD also proves to be effective in testing, as shown by various studies (Silva and Fitzgerald, 2021) and the numerous open-source or commercial tools available. In this project, we have used the *Behave* library[3].

The natural language used to describe these scenarios is known as *Gherkin* (dos Santos and Vilain, 2018). An example of a test written in this language and methodology can be found in Listing 1. Note that tests can be written both before the implementation of the functions and after the implementation of the functions or part of them to evaluate the interaction of the components according to some requirements.

The use of behavior-driven development (BDD) and natural language for writing tests in the game development industry (where our evaluation took place) is driven by two key factors:

1. A significant proportion of game development staff (e.g. QA, designers, producers, etc.) lack programming experience. Therefore, the use of a natural language with a comprehensive library of possible test descriptions is advantageous.
2. Tests must be reusable. The guide library with the components of the test description language acts as a bridge between the development team that provides the required test functions and the members who implement them.

The example in Listing 1 shows only some features of the tests defined in BDD and Gherkin. The tag keyword, such as *@Physics*, specifies a category for the test. It is important to group the tests into categories for two reasons: a) to have a general filter for the AI agents and b) for the sampling methods that decide in which direction more or less computational resources should be spent on the tests. The general pattern of a test specification is to define three main steps:

1. Set the context of the application by using the *Given* step. In our example, the state of the application to be tested must have a started game instance for the test to be executed. More complex examples that use a similar context can be specified with the *Background* or *Outline* keywords to avoid repeating the same contextual setup.
2. Set when the test should start with *When* step. This is the trigger for starting the test. In this example, we start the test when certain conditions occur, as shown in the table in Listing 1. A test can be valid multiple times during runtime.
3. Expected test outcome. Specifies the correct expected results of the test with the keyword *Then*.

Each of the three main steps can contain complex conditions, which are defined in the respective test description with logical keywords such as *Or*, *And*, or

[3]https://github.com/behave/behave

```
Feature: @Physics Simulate a simple car in a racing game. You should consider
    engine power, aerodynamics, rolling resistance, grip and braking force. For
    simplicity, the engine will provide constant power for the game
Scenario Outline: The car should accelerate to the target speed within some
    expected time range.
Given:   We start a game instance on default test map AND
      the car has <power> kw, weights <weight> kg, has a drag coefficient of <drag
    >
When:    I accelerate to reach 100 km/h
Then:    the time should be within a range of 0.5s around <time>s


    Examples:
    | power   | weight    | drag    | time   | name
    | 90      | 1251      | 0.38    | 6.1    | Buzz
    | 310     | 2112      | 0.24    | 3.9    | Olaf
```

Listing 1: An example of a BDD test written using Gherkin syntax. The specification is given in natural language, but each step definition is closely linked to the implementation (Figure 1). The test first creates a context with a new game on a test map and then deploys a car with given parameters on a straight road. The car is tested against the time it takes to accelerate to 100 km/h. This requires complex physics integration mechanisms in the backend that represent the test targets. To reuse the same test across multiple instances, tables are used in the test methodology, e.g. in the case shown there are two test cases with the same template. The role of the AI agent and the LLM is to break the tight coupling of the step definitions to the source code base and allow for a natural language instruction with possible syntax or grammar errors, different parameter sequences or naming, as shown in Figure 3.

```python
@given("We start a game instance on default test map")
def step_impl(context):
    context.car = ObjLib.GetCar()
    context.map =ObjLib.LoadMap("test_map")

@given("the car has (?P<engine_power>\d+) kw, weights (?P<weight>\d+) kg,"
       "has a drag coefficient of (?P<drag>\d+)")
def step_impl(context, engine_power, weight, drag):
    context.car.engine_power = float(engine_power)
    context.car.weight = float(weight)
    context.car.drag = float(drag)
@when("I accelerate to (?P<speed>\d+) km/h")
def step_impl(context, speed):
    speed_in_ms = float(speed) / 3.6
    context.car.set_power(100)
    while context.car.speed < speed_in_ms:
        context.car.simulate(0.1)
@then("the time should be within a range of 0.5s of (?P<time>\d+)s")
def step_impl(context, precision, time):
    assert_that(context.car.time, close_to(float(time), float(precision)))
```

Figure 1: A snapshot from one of the steps implementation source code in the project where the testing case from Listing 1 is supposed to run, showing the coupling between the step definitions and implementation. The step implementation file in the example connects further to exposed functionality implemented by development side (e.g., *ObjLib, Car, Map* and their related functionalities. Many well known programming languages can be used to expose already written functionalities to Python through interoperability libraries.

But. The implementation library for the leaf calls of the steps is created by the developers in a different source code file to hide implementation details that other parties are not interested in. In the example in Listing 1, a data table is used to reuse the same test for a parameterized set of contexts, triggers and expected results. These parameters become input arguments for the source code of the test implementation. More examples can be found in our repository.

4 METHODS

Overview. The interaction between user and assistant is handled via a user interface (UI) based on Streamlit[4], which integrates a routing agent (Smith et al., 2023), an LLM and a human-in-the-loop mechanism, Figure 2 provides a detailed technical overview. This integration aims to improve the adaptability of the system and the efficiency of decision making. The reasoning process follows the work in (Yao et al., 2022), where a ReAcT-type agent and NLP techniques are used to consider the current context (test state and user request) and then select an action that can be executed.

4.1 Functionality

The method we propose works in the following way. The assistant performs the creation of the test step by step, with the user in full control: rewinding, editing steps, suggesting a new input for function call parameters, changing some commands, etc. A concrete example can be found in Figure 3, where the next logical step to be generated would be a *Given* type. The user, who does not know the source code implementation or the available functions of the project, wants to describe his request in natural language. The assistant

[4]https://streamlit.io/

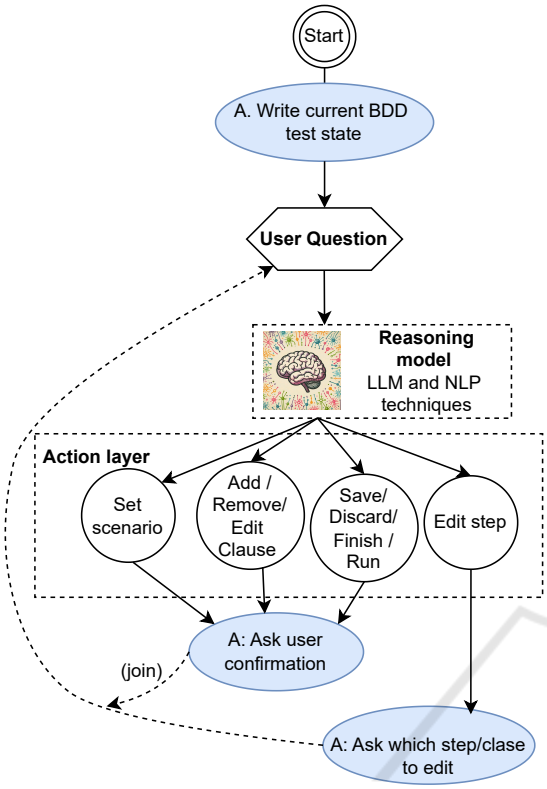

Figure 2: The overview of the agent architecture that relays messages between reasoning with the fine-tuned LLM model, acting and processing user interventions. The blue colored circles represent messages displayed by the agent in the UI application. The hexagon represents the user interface with which the user can react and interact. The inferential NLP and LLM-based model performs the decision-making. Depending on the result, the processed information and decisions are forwarded to the subgraphs, which may contain one or more tool calls (functions) implemented in the framework. For example, if the model decides that the user wants to edit a certain step, it goes through the process by first asking which step should be edited, then the content that should be changed in the step description, and so on. At each point, the model can ask for further explanation or cancel the previous request if it is not understandable.

will then try to process the request, splitting it into logical parts as an abstract syntax tree connected to the source code in the form of function calls and their corresponding parameters.

More recent studies, such as (Karpurapu et al., 2024), have shown that LLMs can generate complete test scenarios with a good acceptance rate if only the user story is used. Intuitively, this is attributed to the fact that LLMs are generally fine-tuned using an important set of data that includes the source code and the tests. However, our experiments have shown that while LLMs can generate acceptable tests, internal knowledge about the project under which the tests are generated can positively influence performance (Section 5). To incorporate this knowledge, we use Retrieval-Augmented Generation (RAG) techniques to control the generation. In this process, the source code is first analyzed to extract the existing tests using a combination of hand-written scripts and reflection support. For a project P, one of the tools within the framework can create the structure shown in Equation 1:

$$Imp_i := (S, CG, CW, CT)_i, \quad Imp_i \in P \qquad (1)$$

Each of the three lists of steps contains details about the functions called, their positions in the source code implementation and parameters (e.g., Figure 3). As motivated by the study in (Storer and Bob, 2019), the semantic extraction of the meaning of each step is done sequentially using two NLP techniques (Mitkov, 2022):

1. Part of Speech Tagging (POS) - which assigns a part of speech to each word in a text
2. Semantic Role Labeling (SLR) - which assigns semantic roles in the form of predicate arguments (e.g., who did what, where and how?).

For each step $S_i \in CG|CW|CT$, a feature vector is created from this information using the library *spaCy*[5]. We refer to this function as $F(S_i)$. Finally, the output is converted into an embedding space (floating numbers) by a sentence conversion model (Reimers and Gurevych, 2019) model, $F_e(S_i)$.

To efficiently store data, support indexing, and manage continuous updates, a vectorized database using the Faiss (Douze et al., 2024) library is used. The similarity of the two steps is determined by a customized cosine similarity function. For a natural language query Ur (e.g. the user's step *Given* in Figure 3) to be matched with the closest step implemented in the project, the same function $F_e(U_r)$ is applied and then evaluated based on the entries in the database using cosine similarity. The variable S represents a string for the scenario description, CG is a list of "Given" steps, CW is a list of "When" steps and CT is a list of "Then" steps.

4.2 The Reasoning Model

When modifying or adding a BDD test step $TS \in \{Given, When, Then\}$, the process takes as input the text for it in natural language, as if for a person who does not know the implementation library or needs to write the parameters in a specific order, and without any need for grammatical correctness. For example, consider the sentence shown in Figure 3:

[5]https://spacy.io/

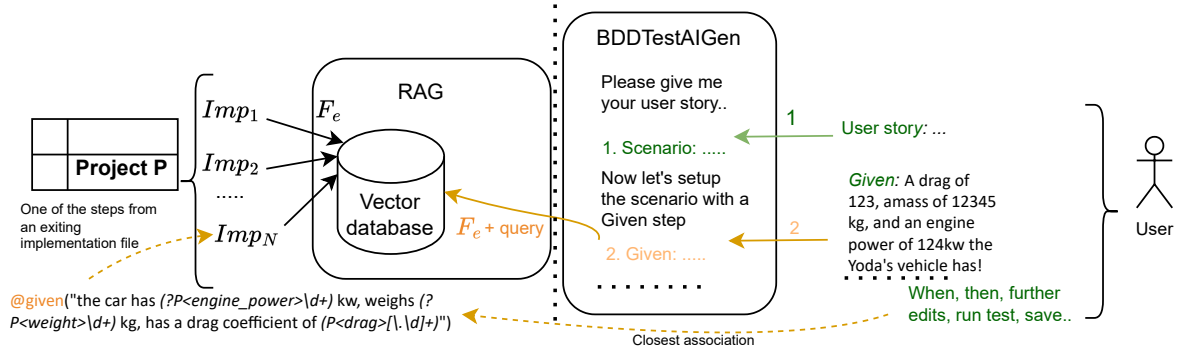

Figure 3: The left-hand side contains the backend services of the framework for processing the functions of the implementations that are made available to the testing side in a project P in the step implementation files. The source code metadata is stored, indexed and queried in response to user requests. The center component represents the *Assistant* component of the framework, which guides the user in completing a full BDD test. On the right-hand side, after the user has provided a user story and the assistant has suggested a scenario, the next logical part is to set up the test with a *Given* step. The user requests this in natural language, with different word order, synonyms and other propositional changes. The request is transformed into the embedding space, queried and the closest pairs of steps in the source code are displayed. The user can confirm, edit or add missing parameters if the assistant cannot assign all functions and parameters correctly.

U_{TS}^{Inp} ="A drag of 123, amass of 12345 kg, and an engine power of 124kw the Yoda's vehicle has!".

The purpose of this process is to map U_{TS}^{Inp} to the exposed functionalities of the project, both in terms of the functions called and the correct parameters. The proposed AI agent, based on LLM and NLP techniques, consists of three main steps:

Step A: Create the feature vector using the F function described above. After using the POS and SRL techniques for feature processing, the output of the user input, U_{TS}^{Inp}, has the representation in Listing 2:

```
F(U_TS^Inp)={[
FullPhrase_0: original raw user input.
 Predicate_0: "has",
 Arguments_0: "the Yoda's vehicle"
 Attributes_0: "A drag of 123",  "a mass
    of 12345 kg", "an engine power of
    124kw"
]}
```

Listing 2: Extraction at feature level using Semantic Role Labeling (SRL) of the example shown in Figure 3.

Step B: Find the closest step that is implemented in the library and matches the text description. Denote the output of the operation with one or more function calls $U_{TS}^{Func} = \{f_1, f_2, ...\}$ based on a complex expression, where each of the functions is a concrete functionality exposed by the implementation. This step is formally described in the Equation 2. The prompt used by the LLM is shown in Listing 3.

$$FindStep(U_{TS}^{Inp}) = Expr(U_{TS}^{F}func) \quad (2)$$

Step C: Match the parameters from the natural language description, U_{TS}^{Inp}, as closely as possible to the parameters required by the functions (e.g. drag=123, mass=12345, engine=124). The metadata of the exposed functions, i.e. the knowledge of the parameter names and their types, f_i^{params}, and a possible comment of the developer f_i^{comm}. Formally, this step is shown in Equation 3. Listing 4 shows the prompt used internally.

$$MatchParams(U_{TS}^{Inp}, F = \{..., f_i^{params}, f_i^{comm}, ...\}) = \{f_i^{p_j \in params} = value\}_{i,j} \quad (3)$$

Human-in-the-loop. Note that there is also the possibility that the AI agent cannot find a corresponding implemented step, is uncertain or does not find all suitable parameters. In this case, the error is reported to the user and they are asked for help to edit the generated step. For example, if the assistant is not sure which step is the correct one or could not find one at all, a graphical user interface shows the user some available options that correspond to the semantically closest step retrieved from the backend LLM. If the parameters are difficult to match, a partial match is displayed along with the missing matches. The user can see the parameters and comments of the corresponding functions in the graphical user interface and then edit the agent's response.

Even if the AI agent is not able to fully map the step or parameters, the display of the obvious options and the fact that both programmers and non-programmers do not have to search large repositories for exposed functionalities could be an advantage for productivity and encourage software testing in general.

Fine-tuning. The method used is completion-only training (Parthasarathy et al., 2024) to save the

```
From the available Gherkin steps listed below, select the one that comes closest
    to the user input step and return it. Do not write any code, just specify the
    step found.
Use the Json syntax for the response. Use the following format if a step can be
    found:
{{
  "found": true,
  "step_found":  the step you found
}}
If no available option is OK, then use:
{{
    "found": false,
}}
Do not provide any other information or examples.

### User input features:
{input_step_to_match_features}
### Available steps features:
{available_steps_features}
```

Listing 3: The template prompt is used to match the input given by the user with the available implementations. There are two variables in the template that are filled at runtime. First, since the type of Gherkin step being added or edited is known, the variable *available_steps_features* contains only the steps that are available for that particular type. The processed user input (e.g. the input of 2) is passed via the variable *input_step_to_match_features*. The few examples have been omitted for reasons of space (available in the repository).

number of tokens injected into the LLM, since the filled prompts can become large, especially when considering small models.

The process begins by assembling pairs of ground truth data consisting of Gherkin step types and all corresponding implemented steps from each project P. For each step type, we ask GPT4 to generate $N = 3$ variation descriptions (keeping the same semantics and number of parameters) that resemble a user's input in natural language. These variations are tested and sampled similarly to Listing 5. The LLM model is then fine-tuned to understand the correspondences between the original and modified versions of the step descriptions.

5 EVALUATION

5.1 Datasets

The dataset used for fine-tuning and evaluation consists of datasets written by experts (programmers) and datasets generated synthetically with GPT4. In the first category, there are 13 open-source GitHub: a) 8 projects already used by (Liu et al., 2024), b) 3 well-documented projects that we found and referenced in the repository, c) 2 private projects with tests for Disney Speedstorm[6] and Asphalt[7] (these have tests written by experts and were analyzed using the computer vision methods defined in (Paduraru et al., 2021)). The numbers of the curated datasets can be found in Table 1.

Table 1: The number of curated tests in our dataset and the number of available steps implemented by each kind.

Feature	Value
Number of available defined tests	259
Number of step implemented by type / total	2,682
Given	1911
When	357
Then	414

5.2 Quantitative Evaluation

To evaluate the methods from this point of view, we use a synthetic generation of correct tests with GPT4, as shown in Listing 5. We measure the integration of the processes of the proposed methods (i.e., feature extraction using NLP techniques, pruning mechanisms, and small-size fine-tuned LLM) by how well they are able to translate the step descriptions obtained by varying the original descriptions into the same source code implementations.

The results are shown in Table 2. First of all, it should be noted that GPT4, which is a much larger model, was able to implement most of the variations correctly. However, the fine-tuned model with a class

[6]https://disneyspeedstorm.com/

[7]https://www.gameloft.com/game/asphalt-8

```
Given the Gherkin step in the target, your job is to extract the parameters and
    assign values to them using the user input.
Do not write any code, but simply specify the values in Json format as in the
    following example
// (Note: The following two examples are simplified for explanatory purposes)
Example:
### Input: The plane has a travel speed of 123 km/h and a length of 500 m
### Target: @given(A plane that has a (?P<speed>\d+) km/h, length (?P<size>\d+) m
Response:
{{
 "speed" : "123 km/h",
 "size" : "500 m"
}}

Another example with template variables (e.g., <variable>) that you need to copy
    in the result as below if used:
### Input: The plane has a travel speed of <speed> km/h and a length of <size> m
### Target: @given(A plane that has a (?P<speed>\d+) km/h, length (?P<size>\d+) m
Response:
{{
 "speed" : <speed>,
 "size" : <size>,
}}

Your task:
### Input:  {user_input_step}
### Target: {target_step}

Response: your response
"""
```

Listing 4: Prompt used to fill in the parameters for the functions called by the step found to match. The variable *user_input_step* is filled with the features of the user input, while *target_step* contains the features of the step found in the previous step, Listing 3. The second example shows the use of parameters with regular patterns, which, as we found out in experiments, are inherently known by the standard LLMs pretraining. Using the example of $(?P < speed > d+)$ and following the Python language representation, *speed* stands for the name of the group to which the match applies, while $d+$ matches a decimal number with one or more digits.

size of 8B, which can be deployed on users' machines, performed well, matching perfectly in more than half of the tests. The remaining errors, broken down by cause, show that the model knows in principle how to combine the step descriptions varied by GPT4 with the original intent (46 failed cases out of 499 tests). These errors are displayed in the GUI, for example, and LLM suggests the closest versions where it is unsure.

The remaining errors were parameters matching, split by either they were wrongly assigned without reporting, or reported as unsure. One important observation at this point is that these errors were mostly caused by the variations of unit measures. A concrete example is the conversion between kW to Hp when used for the engine power. The implementation functions were expecting the value in kW, but GPT4 created variations of values with Hp, which it was able to understand from the parameter names or/and comments. The small-sized model was not able to perform the conversion ($1Hp \sim 0.7457kW$). Fine-tuning explicitly for the conversion of types and units could be addressed in future work to improve the results. The errors observed suggest that the model can be useful, but user confirmation is still necessary, Figure 2 since defects produced by different orders of parameter values without compilation errors could be a difficult problem to debug for example.

5.3 Qualitative Evaluation

From this perspective, we tried to assess the extent to which the AI assistant helps both technical and non-technical people to create BDD tests. The evaluation took place among the developers of the two announced games, more specifically 7 designers and artists (non-technical) and 4 software engineers, who were asked to write 10 tests each in areas where they should know the high-level design of the product.

In the first group, users were able to write tests, taking an average of 4.7 iterations for each step (by an iteration in this case, we mean writing an input, al-

Table 2: Summary of the evaluation performed on the synthetical generated dataset using GPT4.

Description	Count
Num GPT4 synthetically generated tests	1983 out of 2590 expected
Num matched by BDDTestGenAI	1484
Failing cases	499
Reason	
Not able to link a step description	46
Not able to match parameters	423
Proposed incorrect order without reporting	87
Reported that it cannot find at least one	336

lowing the AI to generate a full step with implementation grounding or report issues, and so on until the user's final approval). Most of the failures and retries were due to the variable names not being named correctly or not being annotated. In principle, this group felt that the assistant helped them a lot, as they did not know the source code and did not know what was behind it, but they knew logically what they wanted to achieve in natural language. In this sense, the assistant guided them in their efforts by linking or suggesting existing function prototypes and parameters that they had no idea about.

Similar feedback was reported for the second group (the technically proficient) with an average of 2.3 iterations for each step. The general conclusion in this group was that the assistant was more akin to a contextual code base search, which could potentially increase productivity and the practice of writing tested software.

Computational effort: During this user-based experiment, we also measured the time it takes the user to create a step from a natural language description. We took into account the typical hardware available to developers in the tested use case, on home computers. The tests were carried out on: a) an NVidia GPU 4090 RTX, b) an AMD Ryzer 7 7840u CPU. In the first hardware configuration, the model achieved 108.5 tokens per second, while in the second case it was around 7.3 tokens per second. The number of tokens varied between the stage types, but the average of tokens used for each input during the experiment was ~ 21. This means that the response on a GPU is almost instantaneous, whereas the user would have to wait around 3 seconds on the CPU.

5.4 Limitations and Technical Challenges

One of the biggest challenges in this work was the question of how to use models with large information contexts. To solve this challenge, even though it is currently only partially solved, we used various filtering and pruning strategies before invoking the LLM. For example, each new test was tagged, e.g. *physics* and *animation*. The source code sub-folders were also tagged (in the two game-related projects) so that the information sent to the LLM is significantly filtered before sending the available implementation steps. By using reflection techniques in programming languages, the available set of functions, parameters, and goals was also condensed to present only what is of interest to the reasoning process of the LLM.

In the course of development, many problems were also raised on the user side, which have since been solved. For example, one useful tool not available in the initial phase was the ability to report that a certain type of functionality was missing from the implementation. Thus, the current GUI version contains a path to suggest a new prototype for a step implementation, using only function names, parameters and a general comment. In the test creation process, an empty implementation is added using the proposed prototype and is then to be completed during the project. In this way, the proposed methods fulfill the *Test-driven development (TDD)* (Beck, 2022) methodology. One important remaining issue is the conversion of values (as mentioned above in this section) from the natural language to the assumed data types for the parameters of functions, which may need further fine-tuning. Another problem we have observed is that without a reasonable naming of functions and parameter values, the LLM has almost no clues to assign them. However, we think that this is a general problem, not for the analyzed problem, but in software development.

Some details are not shown in the subgraphs of Figure 2 for space reasons. It was necessary to add small low-level tools that the LLM needed to solve the user requests. For example, a user input needed to be parsed to find file names (save/execute/load), check the paths for correctness and then confirm/notify the user if a problem was found. Without the middle layer provided by the implemented tools that LLM is aware of, it was difficult or nearly impossible for the fine-tuned model to understand some of the user input in this BDD test area.

```
for each test t in Dataset:
 Ask GPT4 to produce 3 variations for each of the three step types
 VT = From a total of 27 possible combinations, select a maximum of 10 tests that
     are still correctly matched and have the same passed status.
 for each sample T in VT:
    Simulate a user requesting the same step descriptions as in T
    Check whether the steps are correctly linked to the same implementations and
    have the same passed status.
```

Listing 5: The pseudocode used to quantitatively assess the correctness of the methods.

During the experiments, we made an important observation that helped the LLM guide with the existing implementation prototypes: The user story description or any hints that the user might say should be defined at the beginning of the process. This is illustrated in Listing 1, in the descriptions *Feature* and *Scenario Outline*.

On a technical level, the framework uses the Huggingface[8] version of the Llama3.1 8B model as a basis for fine-tuning and LangChain Agents[9] for agent-based AI as implementation libraries. For writing BDD tests, the Gherkin[10] language syntax and PyBehave[11] as a framework were preferred.

6 CONCLUSION

This paper presents a method that combines BDD testing techniques in software development with the latest contributions in the field of AI agents and LLMs. An open-source framework is provided for further experimentation for both academia and industry. The experiments conducted so far indicate that AI-assisted generation of tests with human-in-the-loop can improve user experience and productivity and enable the adoption of testing methods for non-technical stakeholders or simplify the process for technical users. Considering the results obtained both synthetically and in experiments with developers, combined with the computational effort required and the ability to run on typical user hardware, we conclude that small, fine-tuned models combined with different processing and pruning strategies can be good enough for both productivity and efficiency.

[8]https://huggingface.co/meta-llama/Llama-3.1-8B

[9]https://www.langchain.com/agents

[10]https://cucumber.io/docs/gherkin

[11]https://pytest-bdd.readthedocs.io

ACKNOWLEDGEMENTS

This research was partially supported by the project "Romanian Hub for Artificial Intelligence - HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021-2027, MySMIS no. 334906 and European Union's Horizon Europe research and innovation programme under grant agreement no. 101070455, project DYNABIC.

REFERENCES

Beck, K. (2022). *Test driven development: By example*. Addison-Wesley Professional.

dos Santos, E. C. and Vilain, P. (2018). Automated acceptance tests as software requirements: An experiment to compare the applicability of fit tables and gherkin language. In Garbajosa, J., Wang, X., and Aguiar, A., editors, *Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May 21-25, 2018, Proceedings*, volume 314 of *Lecture Notes in Business Information Processing*, pages 104–119. Springer.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. (2024). The faiss library. *arXiv preprint arXiv:2401.08281*.

Garousi, V., Bauer, S., and Felderer, M. (2020). Nlp-assisted software testing: A systematic mapping of the literature. *Information and Software Technology*, 126:106321.

Grattafiori, A. et al. (2024). The llama 3 herd of models.

Irshad, M., Britto, R., and Petersen, K. (2021). Adapting behavior driven development (BDD) for large-scale software systems. *J. Syst. Softw.*, 177:110944.

Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., and Narayanan, A. (2024). Ai agents that matter.

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*, 12:58715–58721.

Li, K. and Yuan, Y. (2024). Large language models as test

case generators: Performance evaluation and enhancement.

Liu, Y., He, H., Han, T., Zhang, X., Liu, M., Tian, J., Zhang, Y., Wang, J., Gao, X., Zhong, T., Pan, Y., Xu, S., Wu, Z., Liu, Z., Zhang, X., Zhang, S., Hu, X., Zhang, T., Qiang, N., Liu, T., and Ge, B. (2024). Understanding llms: A comprehensive overview from training to inference.

Lv, K. et al. (2024). Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8187–8198. Association for Computational Linguistics (ACL).

Mitkov, R. (2022). *The Oxford Handbook of Computational Linguistics*. Oxford University Press.

OpenAI et al. (2024). Gpt-4 technical report.

Ouédraogo, W. C. et al. (2024). Large-scale, independent and comprehensive study of the power of llms for test case generation.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2021). Automated game testing using computer vision methods. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 65–72.

Parthasarathy, V. B., Zafar, A., Khan, A., and Shahid, A. (2024). The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992. Association for Computational Linguistics.

Shrestha, A. et al. (2025). A survey and insights on modern game development processes for software engineering education. In *Software and Data Engineering*, pages 65–84. Springer Nature.

Silva, T. R. and Fitzgerald, B. (2021). Empirical findings on BDD story parsing to support consistency assurance between requirements and artifacts. In Chitchyan, R., Li, J., Weber, B., and Yue, T., editors, *EASE 2021: Evaluation and Assessment in Software Engineering, Trondheim, Norway, June 21-24, 2021*, pages 266–271. ACM.

Smart, J. F. and Molak, J. (2023). *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster.

Smith, J., Doe, A., and Lee, B. (2023). From llms to llm-based agents for software engineering: A survey of current challenges and future. *Journal of Software Engineering*, 25(4):123–145.

Storer, T. and Bob, R. (2019). Behave nicely! automatic generation of code for behaviour driven development test suites. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 228–237.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.