AutomTest 3.0: An automated test-case generation tool from User Story processing powered with LLMs

Joanderson Gonçalves Santos
joanderson.goncalves@ufba.br
Computer Science Department
Federal University of Bahia - UFBA
Salvador, Bahia, Brasil

Rita Suzana Pitangueira Maciel
rita.suzana@ufba.br
Computer Science Department
Federal University of Bahia - UFBA
Salvador, Bahia, Brasil

ABSTRACT

Test-driven development (TDD) and Behavior-driven development (BDD) approaches address software development issues, proposing the creation of unit tests for each system component before implementing the code for the corresponding functionality. This approach results in an objective and compact solution. It ensures that the entire codebase is tested and enhances the overall quality of the software. However, the testing process can be time-consuming and resource-intensive. Some tools were proposed to address those issues, but they lack functionalities based on User Story in a TDD approach. In this context, we propose the AutomTest 3.0 tool, which facilitates unit test case generation from user stories prior to writing the code base related to the functionality. Built upon previous versions of AutomTest, this tool extends the other ones by including generative AI to power the creation of suggestions for methods the software under test should have. This tool creates test cases for Java using techniques such as equivalence class partitioning, natural language processing, and generative artificial intelligence. To evaluate the effectiveness of AutomTest 3.0, we conducted an exploratory study with software development professionals. Their feedback highlighted the tool's utility in their daily work routines. AutomTest 3.0 demonstrated promising results in test case generation, scenario coverage, and speed advantages in test case creation. https://youtu.be/xoTrvhlfvu8

KEYWORDS

Test Case Generation, Test-Driven Development (TDD), Natural Language Processing (NLP), Generative AI, User Story

1 INTRODUCTION

Software requirements are necessary for addressing real-world issues including activities automation and business process support [1]. User stories describe desired user functionalities, being a useful feature that includes textual explanations, discussions, and testing for confirmation [4].

Similarly to requirements-related activities, testing is essential in the software development life cycle. Software testing aims to identify defects, errors, or missing requirements, providing stakeholders with accurate knowledge about product quality [9].

To further enhance software quality and ensure robust code implementation, the Test-Driven Development (TDD) approach is often employed. According to [12], Test-Driven Development (TDD) advocates that the practice of creating tests before code implementation not only ensures comprehensive test coverage but also promotes a clear understanding of the desired requirements and functionalities [12]. Moreover, an incremental testing approach leads to code quality improvement and a cleaner design [18] which ultimately results in more reliable and maintainable software [12][18].

Furthermore, Large Language Model (LLM) strategies are usually associated with transformer-based models with billions of parameters, trained on trillions of tokens [17]. These LLMs serve as response generators and can perform various language-related tasks accurately, with significant advancements in recent years for distinct purposes[11].

Numerous automated test-generation tools powered with LLM were proposed to assist developers in writing tests, aiming to increase coverage and generate exploratory inputs. Despite the many tools, they often offer implementation techniques dependent on an already developed code base, which implies an inadequacy to cope with test-driven development needs, such as in [16][3][10].

Intended to aid in software development and be compatible with a TDD approach, the initial version of AutomTest [7] was built to help automatically generate tests for Java-developed software, using the tester's expertise without relying on programming abilities. The JUnit framework is employed to execute the test cases created by the tool for the Java programming language using the combination of equivalence class partitioning techniques. AutomTest's second version added Natural Language Processing (NLP) from User Stories, expanding on the features of the first version [5].

This paper presents the third edition of AutomTest, an extension of version 2.0 that offers enhanced user experience and expanded language support to accept User Stories in English, in addition to the Portuguese language support the previous versions provided. Furthermore, AutomTest 3.0 exploits two LLMs, Google's Palm [1] API and OpenAI's GPT-3.5-Turbo [2], to power User Story data extraction, to provide better results regarding the number of method signatures and returns suggested and their completeness concerning data type definition.

To evaluate Automtest 3.0, we conducted an exploratory study with five software development professionals. They used AutomTest 3.0 and evaluated the tool through a questionnaire regarding easiness of use, overall user experience, and relevance in the software development area. Overall, the tool was perceived as having a potential utility within the software development area, offering time-saving possibilities while developing system functionalities with well-defined user stories.

This paper is structured as follows. Section 2 discusses using Large Language Models with testing tools and strategies, presenting some related works. Section 3 dives into the AutomTest 3.0

[1]https://ai.google.dev/palm_docs/
[2]https://platform.openai.com/docs/models/gpt-3-5-turbo/

functionality, offering a view of the tool's capabilities. Section 4 presents the tool's validation process and results. Section 5 presents the results obtained from the validation process while in section 6 AutomTest's limitations are introduced. Finally, section 7 discusses the results obtained and offers possible future work.

2 LARGE LANGUAGE MODELS AND TESTING

Considering the impressive performance on several tasks related to NLP and programming [2], different NLP-related studies [14][13][8] and LLM-powered test automation tools have been proposed [16][3][10]. While this combination is promising, the field is still at an early stage and requires further investigation [14].

A user acceptance testing framework based on user stories and acceptance criteria is presented in [13], essentially for manual testing. This approach creates a test case diagram linking the affected user to the relevant functionality. The authors recommend using this framework to generate comprehensive acceptance test cases in natural language, based on acceptance criteria. This would enable the evaluation of acceptance criteria fulfillment in percentage terms.

In [8], Güneş and Aydemir address the issue of user stories' simple structure hindering the capture of relationships between them, which complicates understanding and organizing backlog items. They propose constructing goal models to establish explicit relationships, using NLP to extract information from user stories. Like AutomTest, the author employs NLP to process user stories, although their result is not a test suite but a set of heuristics for generating goal models from user stories, leveraging NLP to produce models comparable to those created manually.

Schafer et al. [16] extensively evaluated LLM's effectiveness, specifically using OpenAI's gpt-3.5-turbo, for automated unit test generation in JavaScript. Their approach involves providing the LLM with prompts that include function signatures, implementations, and usage examples from documentation. The result demonstrates the feasibility of automation in unit test generation.

In [10] Kang et al. propose the framework LIBRO (LLM Induced Bug ReprOduction), a framework that uses LLMs to generate tests, process the results, and suggest solutions from general bug reports.

The ChatUniTest [3] is an automated unit test generation tool based on ChatGPT developed within the Generation-Validation-Repair framework. ChatUniTest analyzes the project, extracts essential information, and creates an adaptive context that includes the focal method and its dependencies within a predefined token limit. This context is incorporated into a prompt and submitted to ChatGPT [3].

All these works share similarities with AutomTest 3.0. As in [8], AutomTest uses NLP to extract data from user stories, and, as in [16][3][10], AutomTest also generates tests using LLM. However, AutomTest generates the test suite from software requirements presented in user stories and, therefore, is a tool adequate to use with the Test-Driven Development (TDD) methodology. The other tools, however, focus on generating tests for already developed code bases, which implies an inherent inadequacy to cope with test development iterations.

3 AUTOMTEST TOOL: A UNIT TEST GENERATOR

Seeking to assist in software development, the AutomTest was created to enable the automated generation of tests for software developed in Java using the tester's perception without requiring programming skills and even by individuals who do not have technical knowledge of coding tests in Java programming language [7].

The tool generates test cases for the Java programming language to be executed using the JUnit framework. It combines equivalence class partitioning techniques and boundary value analysis, which are particularly effective in finding software failures [6].

To use AutomTest, the user provides the tool with the data from the software specification to be developed and receives as output an executable Java code containing a suite of tests based on the software's functional requirements.

3.1 Overview of previous versions

Automtest is currently in its third version. Some functionalities of AutomTest have existed since earlier versions, while others are new additions in the third version presented in this paper.

In its initial version, AutomTest supported software development by creating test cases based on the information provided for each equivalence class. However, the test method input is manually entered by the user. Likewise, the main contribution proposed by AutomTest's second version is information extraction from User Stories using NLP to automatically compose method signatures compatible with the equivalence class partitioning technique [5]. That enhancement leads to a reduction in the time and effort required to use the tool.

Although AutomTest 2.0 made significant advances with NLP, it had limitations in suggesting data types for method parameters and return types, which implies more manual input before generating test suites. Additionally, the previous version could not process all types of user stories, such as listings, updates, and user stories for exclusion processes.

Additionally, during the validation process of AutomTest 2.0, [5] highlighted some suggestions for improvement from participants. Firstly, participants suggested enhancing the method selection interface to make the information-filling process more intuitive. Secondly, they recommended allowing the manual addition of new methods and parameters in the interface that displays the methods extracted from the user story. Finally, there was a suggestion to make the creation of equivalence classes more intuitive.

3.2 AutomTest 3.0

An important new feature the tool presented by this paper introduces is the integration of LLMs with the Automtest enabling automatic retrieval of additional information for method suggestions. The integration with LLMs allows the tool to suggest parameters and method return data types, reducing the need for manual input and increasing application efficiency. Another limitation of previous versions that AutomTest 3.0 with LLM overcomes is the generation of methods for different user story types, not limited to creation-related user stories.

Another improvement provided by AutomTest 3.0 is the multi-language support, English and Portuguese. Moreover, to address the user experience improvement suggestions made by AutomTest 2.0 testers, the tool's third version presented by this paper rebuilds the Graphic User Interface with a larger display, help texts, enhancements in the equivalence class definition, and better control over the methods suggestions and equivalence class creations.

3.3 Navigating AutomTest 3.0

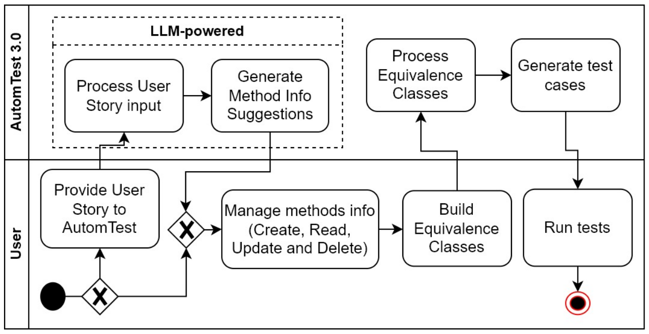
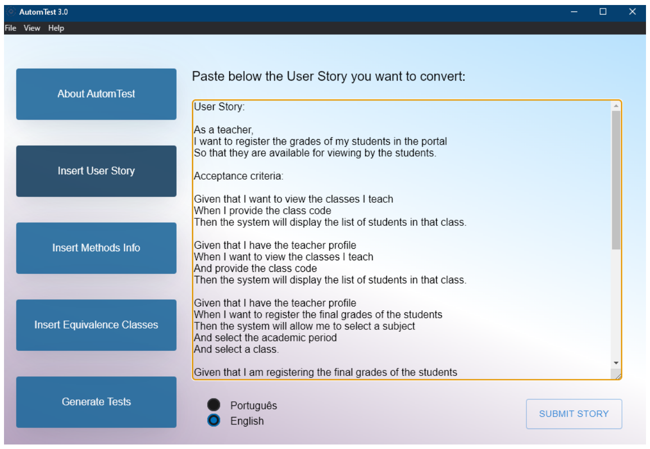

AutomTest 3.0
User
LLM-powered
Process User Story input
Generate Method Info Suggestions
Process Equivalence Classes
Generate test cases
Provide User Story to AutomTest
Manage methods info (Create, Read, Update and Delete)
Build Equivalence Classes
Run tests

Figure 1: Activity diagram corresponding to the test generation process using AutomTest 3.0

The core functionality of AutomTest 3.0 is the automated generation of a Java unit test suite from a user story, with the processing aided by employing LLMs. To use AutomTest, the user must follow the depicted process in Figure 1. As starting input, users are presented with two possibilities: (i) providing a user story as input or (ii) inserting methods information manually.



Paste below the User Story you want to convert:

User Story:

As a teacher,
I want to register the grades of my students in the portal
So that they are available for viewing by the students.

Acceptance criteria:

Given that I want to view the classes I teach
When I provide the class code
Then the system will display the list of students in that class.

Given that I have the teacher profile
When I want to view the classes I teach
And provide the class code
Then the system will display the list of students in that class.

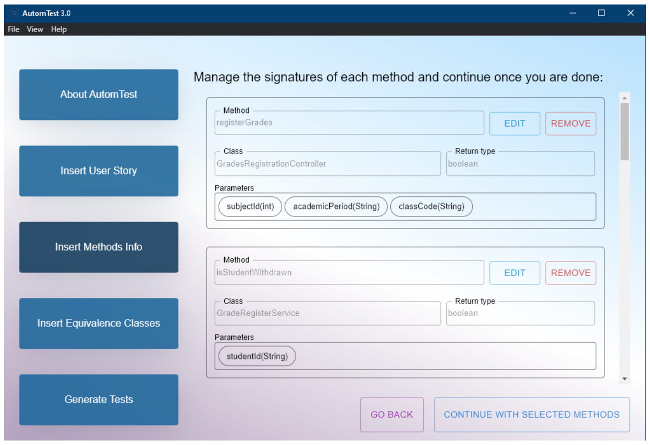
Given that I have the teacher profile
When I want to register the final grades of the students
Then the system will allow me to select a subject
And select the academic period
And select a class.

Given that I am registering the final grades of the students

Português [✓ English]

Figure 2: User Story screen of AutomTest 3.0

One of the possibilities is to start by inputting a user story, enhanced with acceptance criteria, into the AutomTest tool, which accepts user stories written in English and Portuguese (Figure 2). Using the user story as input, AutomTest leverages LLMs' NLP capabilities to extract relevant methods directly, which the system under test will likely need to meet the user story specification and acceptance criteria. The user can then choose which suggested methods are useful and discard any unwanted suggestions. This technique allows LLMs to generate test cases and method suggestions for the system under test by effectively processing the provided user story.



Manage the signatures of each method and continue once you are done:

Method: registerGrades
Class: GradesRegistrationController
Return type: boolean
Parameters: subjectId(int) academicPeriod(String) classCode(String)

Method: isStudentWithdrawn
Class: GradeRegisterService
Return type: boolean
Parameters: studentId(String)

Figure 3: Methods Information list screen of AutomTest 3.0

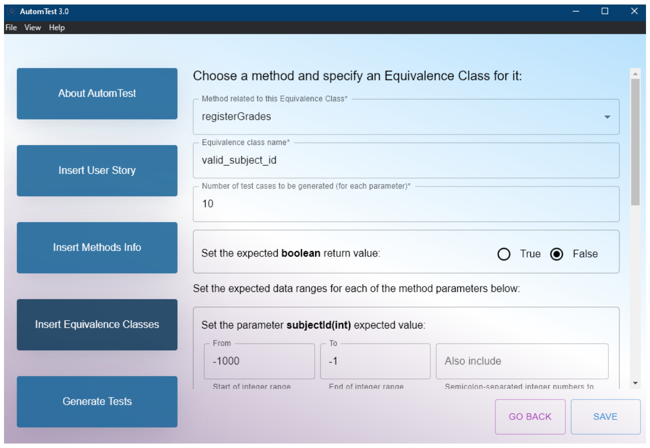
By providing method information directly instead of using a user story, the application user must study the system under test specification to provide AutomTest 3.0 with some information about each method the system is likely to have for the functionality to be tested. The required information on each method is the class name, method name, parameter names, their data types, plus the return data type.

Concerning AutomTest's input possibilities, choosing either way it is possible to create or edit the methods the AutomTest 3.0 processing generated. Either by user story or manual definition, the two possibilities to start using the tool will result in method information being added to the application. Those methods informations can be managed by AutomTest 3.0. It is possible to create or delete method information in the application. It is also possible to edit a method information definition by pressing the Edit button, shown in Figure 3.

Regarding the method parameter possibilities, AutomTest 3.0 accepts parameters with the following types: int, double, float, boolean, char, String, and Date. If any parameter field is incorrectly filled, AutomTest checks and alerts before proceeding. This validation behavior is employed in every AutomTest screen.

The next step then is to define the equivalence partitioning. An equivalence partition refers to a subset of possible values for a variable or set of variables in a test case, where all values within the partition are treated similarly. In other words, the values within the partition are considered "equivalent" in how they are handled. This approach is relevant in software testing, aiming to ensure that the system reacts consistently to different sets of inputs considered equivalent. Each method can have multiple equivalence classes. Figure 4 shows the creation of an equivalence class.

Once the user inputs all the relevant equivalence classes, it is possible to generate the test suite, choosing the location where the Java test classes will be created, with a runnable test code.



Choose a method and specify an Equivalence Class for it:

Method related to this Equivalence Class*: registerGrades

Equivalence class name*: valid_subject_id

Number of test cases to be generated (for each parameter)*: 10

Set the expected **boolean** return value: True [✓ False]

Set the expected data ranges for each of the method parameters below:

Set the parameter **subjectId(int)** expected value:

From: -1000 (Start of integer range)

To: -1 (End of integer range)

Also include (Semicolon-separated integer numbers to

Figure 4: Equivalence class creation screen of AutomTest 3.0

Additionally, AutomTest always verifies the specified boundaries, as boundary values are more likely to contain errors [6].

AutomTest 3.0 can be used in the TDD approach by providing developers with the test suite for the functionality under development. To achieve that the user can provide the tool with a user story depicting the expected functionality and acceptance criteria, or the user can directly provide the expected methods information soon-to-be developed. With that approach, the user can easily generate a unit test suite for the functionality about to be constructed without the need for manual construction of each unit test.

3.4 AutomTest 3.0 Architecture

Written in Python 3, React, and Electron, AutomTest 3.0 is a software built upon the evolution of AutomTest 2.0 (available at [15]) under the MIT License). The tool's source code is available at [3] and [4], front-end and back-end, respectively. Separated in Models, Views, and Controllers, the software architecture was designed using the MVC pattern, focusing on separating concerns.

The application front-end is developed using React, a JavaScript library for building user interfaces, and Electron, which allows for creating cross-platform desktop applications using web technologies. The React components render the user interface and handle user interactions, serving as the View in the MVC architecture. Electron enables the application to run as a desktop application, providing access to native operating system functionalities.

Python 3 was used to implement the back-end. It is accountable for all the logic of user story conversion, with calls to LLM APIs, as depicted in Figure 5. Furthermore, the back-end is responsible for the equivalence classes processing, and Java test case generation, both happening in the generator component, shown in Figure 5's left-hand side.

The front-end and back-end communicate through RESTful APIs. Figure 5 shows the AutomTest API at the top of the back-end component. The React front-end sends HTTP requests to the Python back-end, which processes these requests and returns the appropriate responses. The back-end also provides services responsible

[3]https://github.com/JoandersonG/AutomTest.git
[4]https://github.com/JoandersonG/automtest3.0-frontend.git

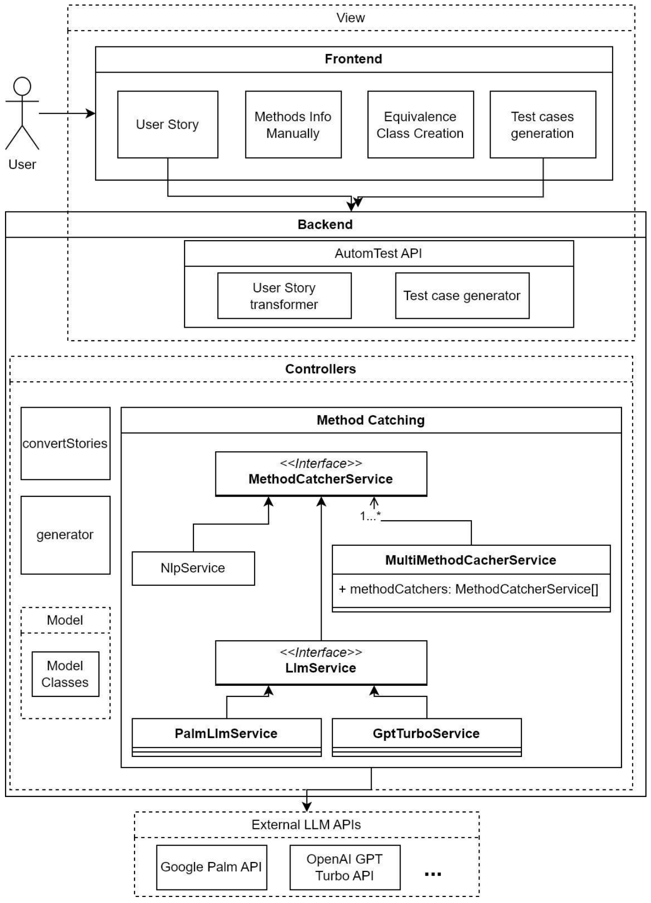


Figure 5: AutomTest 3.0's architecture

for integrating with external APIs allowing communication with LLMs.

The Controllers are responsible for handling HTTP requests and the model classes are used to model the business logic, with equivalence classes, methods, and parameters.

The method caching component depicted in Figure 5 relates as a part of user story processing logic, encompassing all of the services for NLP and LLM API calls. The *NlpService* uses SpaCy[5], an open-source NLP library for Python.

The external LLM APIs currently in use in AutomTest 3.0 are Google's Palm API and OpenAI's GPT-3.5-Turbo API to power User Story data extraction (external LLM API component, bottom of Figure 5). The *PalmLlmService* and the *GptTurboService* were created specifically to handle the API requests for each of the LLMs and are called after a user story input and subsequent API call to AutomTest's back-end.

As the project was designed, there was a preoccupation to keep the LLM integration services loosely coupled with the rest of the application. That concern guarantees the easy evolution and adaptation of AutomTest 3.0 to include or replace the available LLMs integrated into the tool without extensive code modification.

[5]https://spacy.io/

4 TOOL EVALUATION

To evaluate the AutomTest 3.0 we performed a pilot study to gather valuable feedback from users to identify potential improvements regarding easiness of use, overall user experience, and relevance in the software development. We conducted the pilot study in four steps with five software development professionals. Firstly, we performed the presentation step, followed by the tool installation. Each participant needed to install and run AutomTest 3.0 locally on their computer. Next, the users had the opportunity to utilize and explore the tool. In the final step, they answered a survey.

Regarding user profiles, two of them work or used to work with software testing, and three individuals work as software developers. Four of them have two to five years of working in software development-related areas, and the fifth participant has less than a year of experience. All of the five professionals have worked with unit testing previously.

The participants were invited by convenience through e-mail to distinct appointments. Initially, they were presented with basic instructions on key aspects handled by AutomTest. Following this introduction, users were provided with executables of the software tool to install and test on their computers.

For the presentation step, we prepared slides as a guide. We introduced software testing, test-driven development, and equivalence class partitioning concepts. Additionally, we gave an introduction to AutomTest 3.0 and its functionalities. These software demonstrations allowed users to experience the tool firsthand and understand its capabilities in a practical setting. Participants could ask questions at any point during the presentation and were encouraged to explore the tool's features, including manual methods definition, user story input, performing test case generation, and interacting with all the functionalities of AutomTest 3.0.

In the user testing step, participants were provided with two specific user stories previously used in AutomTest 2.0's validation. The user stories included detailed scenarios and tasks that the users needed to perform, in the form of acceptance criteria in the format given-when-then allowing for a structured and comprehensive evaluation of the tool. The participants were assigned to generate a test for the second user story without the assistance of the researcher who conducted the pilot study. This step was crucial in assessing how effectively users could utilize AutomTest 3.0 to generate test cases. After the testing step, users were asked to complete a survey with questions designed to capture users' experiences, identify any issues encountered, and gather suggestions for future improvements.

Users interacted with AutomTest 3.0 by first inputting the provided user story into the tool. The tool then generated a set of methods based on the user story. Users analyzed these generated methods, selecting those that were most relevant to the scenarios described in the user story. Next, they created equivalence classes to categorize and organize possible input/output sets for testing. This process allowed users to structure their tests systematically and ensure comprehensive coverage of possible input variations.

Finally, users analyzed the test cases generated by AutomTest 3.0 based on their input, equivalence classes, and expectations. They assessed the accuracy and completeness of the test cases, comparing them to the expectations set by the user story and their testing objectives. As a final step, users were conducted to a survey[6], in an online environment using Google Meet.

5 RESULTS

Overall, the pilot study was instrumental in highlighting the strengths of AutomTest 3.0 and identifying areas for improvement. The direct interaction with the tool enabled users to provide informed and constructive feedback.

Both user stories provided, available at [7], were successfully employed by all the users of AutomTest 3.0 during the pilot study. Key observations and insights, gathered from the user testing sessions, provided valuable feedback for further development, including the usefulness of being able to edit methods information, in comments such as "I found it very useful to be able to edit parameters, classes, methods, and return types.; " and "It's useful to be able to edit and create methods regardless of the user story".

Some challenges arose and were noticed during the software demonstrations. The main point is the difficulty users non-familiarized with equivalence classes encountered when understanding its use. One user encountered difficulties with the installation process due to varying system configurations. These issues were addressed promptly through troubleshooting sessions, ensuring all users could proceed with their testing.

Users found the interface to be generally user-friendly, although some suggested improvements enhancing the guidance provided during the creation of string patterns for equivalence classes, highlighting the need for more intuitive support features.

The functionality of the tool was highly appreciated, with users noting the efficiency of the method information generation process. The methods produced were generally useful for their testing needs, demonstrating the tool's capability to effectively translate user stories into relevant test components. This positive feedback underscored the practical utility of AutomTest 3.0 in real-world testing scenarios.

Regarding performance, the tool handled the processing of user stories and the generation of test cases well. However, a few users experienced longer processing times due to extended calls to LLM APIs. This indicated a need for further optimization, particularly in parallelizing back-end tasks to enhance performance and reduce latency.

Relevance was another strong point noted by users. The test cases generated by AutomTest 3.0 were deemed useful and straightforward, suggesting that the tool could be effectively integrated into their work environment. This feedback affirmed the tool's potential to improve testing processes and its applicability in professional settings.

Overall, this study pointed to the tool feasibility and these insights were crucial for identifying both the strengths and areas for improvement in AutomTest 3.0. The feedback gathered from user testing sessions provided a foundation for making targeted enhancements to the tool, ensuring it better meets the needs of its users in future iterations and indicating it's a useful tool for software development workers.

[6]https://docs.google.com/document/d/1R5lphUtaIFxWMZmhAUua3RvQMK-RAYjzjALqk10mg7I
[7]https://drive.google.com/file/d/1zC8r8Rl-s5wbTlQRkqafnVKKA1HftZ4g/

5.1 AutomTest 3.0's Limitations

Although the evolution of the AutomTest 3.0 tool has resulted in improvements, certain constraints were identified during the development and testing phases. For instance, the method suggestions provided by AutomTest 3.0 are contingent upon the LLM's capacity to process the user story, which in turn necessitates calls to the LLM's APIs, ultimately resulting in delays in the application's response to the user's input. Additionally, AutomTest 3.0 requires manual input of equivalence classes, which represents a further limitation. Furthermore, the tool is only compatible with Java test generation, which precludes testing generation for other programming languages. These limitations underscore the necessity for further improvement and highlight the challenges to be addressed.

6 CONCLUSIONS AND FUTURE WORK

This paper presented the evolution of the AutomTest 2.0 test case generator tool, incorporating the capabilities of large language models (LLMs) to enhance its functionality. The AutomTest 3.0, now integrating LLMs for NLP, aims to generate test cases with less manual intervention.

The results of this research project indicate that the integration of LLMs into AutomTest 2.0 enhances its ability to understand and process user stories, leading to the generation of test cases with less need for manual intervention, as AutomTest 3.0 can suggest methods with their associate data types included, without the need for manual input on that information.

As future work, some enhancements can be considered to further improve AutomTest 3.0. Something noticed during the validation steps was the time AutomTest takes to process the user story, and some work can be done to parallelize the LLM API calls to achieve better performance. Besides that, studying the integration of large language models within the equivalence class creation step could leverage automation, help reduce manual intervention, and help provide faster results for users. Finally, expanding language support beyond Java is also something that could enhance utility for a larger audience. Moreover, we intend to perform a regular exploratory study considering threats to validity.

REFERENCES

[1] Pierre Bourque and R.E. Fairley. 2014. *Guide to the Software Engineering Body of Knowledge - SWEBOK V3.0.* IEEE and IEEE Computer Society, Washington, DC, United States.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, and et al. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[3] Yinghao Chen and et al. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. ACM, New York, NY, USA, 572–576. https://doi.org/10.1145/3663529.3663801

[4] Mike Cohn. 2004. Advantages of User Stories for Requirements. InformIT Network. Available at: http://www.informit.com/articles.

[5] Venâncio de Sá Oliveira. 2024. AutomTest 2.0: Ferramenta de processamento de User Stories para geração de casos de testes automatizados. Undergraduate Thesis – Federal University of Bahia, Salvador, Brasil..

[6] Narayan Debnath, Adam Kruger, and Melinda Alexander. 2013. A Boundary Value Analysis Tool - Design and Description, In ITNG '13: Proceedings of the 2013 10th International Conference on Information Technology: New Generations. *Proceedings of the 2013 10th International Conference on Information Technology: New Generations, ITNG 2013* 1, 1, 77–82. https://doi.org/10.1109/ITNG.2013.20

[7] Daniel Fernandes and Rita Maciel. 2020. Towards a Test Case Generation Tool Based on Functional Requirements. In *Anais do XIX Simpósio Brasileiro de Qualidade de Software* (São Luiz do Maranhão). SBC, Porto Alegre, RS, Brasil, 386–391. https://sol.sbc.org.br/index.php/sbqs/article/view/14236

[8] Tuğçe Güneş and Fatma Başak Aydemir. 2020. Automated Goal Model Extraction from User Stories Using NLP. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*. IEEE and IEEE Computer Society, Washington, DC, United States, 382–387. https://doi.org/10.1109/RE48521.2020.00052

[9] Muhammad Abid Jamil and et al. [n. d.]. Software Testing Techniques: A Literature Review. ([n. d.]), 177–182. https://doi.org/10.1109/ICT4M.2016.045

[10] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[11] Enkelejda Kasneci, Kathrin Sessler, and et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274. https://doi.org/10.1016/j.lindif.2023.102274

[12] E.M. Maximilien and L. Williams. 2003. Assessing Test-Driven Development at IBM. (2003), 564–569.

[13] Pallavi Pandit and Swati Tahiliani. 2015. AgileUAT: A Framework for User Acceptance Testing based on User Stories and Acceptance Criteria. *International Journal of Computer Applications* 120 (06 2015), 16–21. https://doi.org/10.5120/21262-3533

[14] Indra Raharjana, Daniel Siahaan, and Chastine Fatichah. 2021. User Stories and Natural Language Processing: A Systematic Literature Review. *IEEE Access* PP (04 2021), 1–1. https://doi.org/10.1109/ACCESS.2021.3070606

[15] V. Sateoli. 2022. AutomTest 2.0 Source Code. https://github.com/vsateoli/AutomTest. Accessed: 2024-07-14.

[16] Max Schafer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (1 jan 2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955 Publisher Copyright: © 1976-2012 IEEE..

[17] Murray Shanahan, Kyle McDonell, and Laria Reynolds. 2023. Role play with large language models. *Nature* 623 (11 2023). https://doi.org/10.1038/s41586-023-06647-8

[18] Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus. 2011. What Do We Know about Test-Driven Development? *Software, IEEE* 27 (01 2011), 16 – 19. https://doi.org/10.1109/MS.2010.152