Large Language Models in Software Engineering: A Focus on Issue Report Classification and User Acceptance Test Generation

Gabriele De Vito[1,†], Luigi Libero Lucio Starace[2,*,†], Sergio Di Martino[2], Filomena Ferrucci[1] and Fabio Palomba[1]

[1]*Università degli Studi di Salerno, Salerno, Italy*
[2]*Università degli Studi di Napoli Federico II, Naples, Italy*

Abstract

In recent years, Large Language Models (LLMs) have emerged as powerful tools capable of understanding and generating natural language text and source code with remarkable proficiency. Leveraging this capability, we are currently investigating the potential of LLMs to streamline software development processes by automating two key tasks: issue report classification and test scenario generation. For issue report classification the challenge lies in accurately categorizing and prioritizing incoming bug reports or feature requests. By employing LLMs, we aim to develop models that can efficiently classify issue reports, facilitating prompt response and resolution by software development teams. Test scenario generation involves the automatic generation of test cases to validate software functionality. In this context, LLMs offer the potential to analyze requirements documents, user stories, or other forms of textual input to automatically generate comprehensive test scenarios, reducing the manual effort required in test case creation. In this paper, we outline our research objectives, methodologies, and anticipated contributions to these topics in the field of software engineering. Through empirical studies and experimentation, we seek to assess the effectiveness and feasibility of integrating LLMs into existing software development workflows. By shedding light on the opportunities and challenges associated with LLMs in software engineering, this paper aims to pave the way for future advancements in this rapidly evolving domain.

Keywords

Large Language Models, Vector Databases, Issue Report Labeling, User Acceptance Test Generation, Software Engineering

1. Introduction

In recent years, the field of software engineering has witnessed a paradigm shift with the emergence of Large Language Models (LLMs), such as OpenAI's GPT (Generative Pre-trained Transformer) series [1] or LlaMA [2]. These advanced Natural Language Processing (NLP) models have demonstrated remarkable capabilities in understanding and generating natural language text and source code, sparking widespread interest in their potential applications across various domains. Among these applications, the introduction of LLMs in software engineering holds significant promise for revolutionizing traditional practices and enhancing the efficiency of software development processes [3].

This paper aims to outline our ongoing research focused on harnessing the power of LLMs for two key tasks in software engineering: issue report classification and test case generation. These tasks represent critical components of the software development lifecycle, with implications for both the quality of software products and the productivity of development teams. By exploiting the capabilities of LLMs, we seek to address challenges inherent in these tasks and explore opportunities for automation and optimization.

Issue report classification is a fundamental aspect of software maintenance and bug tracking, involving the categorization and prioritization of incoming issue reports, such as bug reports or feature requests [4]. Traditionally, this process has relied heavily on manual intervention, leading to bottlenecks in response time and resource allocation. Through our research, we aim to develop and evaluate LLM-based approaches for automating issue report classification, with the goal of improving the efficiency and accuracy of this critical task.

User Acceptance Test (UAT) generation is another area of focus in our research, where the objective is to automatically generate test cases that comprehensively validate software functionality. Manual creation of test cases can be time-consuming and error-prone, especially in complex software systems with numerous features and dependencies. By leveraging LLMs, we aim to explore methods for automatically generating test cases from tex-

Ital-IA 2024: 4th National Conference on Artificial Intelligence, organized by CINI, May 29-30, 2024, Naples, Italy
*Corresponding author.
†These authors contributed equally.
gadevito@unisa.it (G. De Vito); luigiliberolucio.starace@unina.it (L. L. L. Starace); sergio.dimartino@unina.it (S. Di Martino); fferrucci@unisa.it (F. Ferrucci); fpalomba@unina.it (F. Palomba)
0000-0002-1153-1566 (G. De Vito); 0000-0001-7945-9014 (L. L. L. Starace); 0000-0002-1019-9004 (S. Di Martino); 0000-0002-0975-8972 (F. Ferrucci); 0000-0001-9337-5116 (F. Palomba)

tual artifacts, such as requirements documents or user cases, thereby streamlining the testing process and reducing manual effort.

The remainder of this paper is structured as follows. In Section 2, we outline the research activities we are currently carrying out in the context of issue report labeling, while in Section 3, we focus on our research on automatic user acceptance test generation. Last, in Section 4, we give closing remarks and outline future works.

2. LLMs for Issue Report Classification

2.1. Problem Description

In collaborative Software Engineering, teams work together to develop and maintain software products. This collaboration involves various stakeholders, including developers, testers, project managers, and end-users, who contribute to different stages of the software development lifecycle. Throughout this process, issue reports play a crucial role in identifying, documenting, and addressing problems or requested changes within the software [5].

Issue reports, which are often managed by dedicated issue-tracking software [6], are formalized descriptions of change requests or issues encountered by stakeholders or identified during testing. These reports typically consist of natural language text written by stakeholders, possibly including details such as the nature of the problem, steps to reproduce it, expected and observed software behaviour, and any relevant screenshots, error messages, or logs. Issue reports serve as a key mean of communication between end-users or stakeholders and the development team, providing essential feedback on the functionality, usability, and performance of the software product.

Issue report classification is a fundamental aspect of software maintenance and bug tracking, involving the categorization and prioritization of incoming issue reports, such as bug reports, feature requests, or documentation-related inquiries [7]. Misclassifying these reports can lead to misallocated resources, delayed bug fixes, and overall inefficiencies in the software development lifecycle. Relying exclusively on manual intervention for this classification task may lead to the introduction of bottlenecks in response time and resource allocation. Moreover, delegating the issue classification task to the stakeholders who submit the issue reports also often results in misclassified reports [8, 4].

2.2. State of the art

Different approaches have been proposed in the literature to address these challenges. Antoniol et al. [9] proposed using machine learning techniques—alternating decision trees, naive Bayes classifiers, and logistic regression—to automatically classify issues in bug tracking systems as either bugs (corrective maintenance) or non-bugs (other activities). The technique achieves classification accuracy between 77% and 82%, highlighting the potential for automated issue routing. However, the proposed approach is limited by its focus on three open-source systems and the manual classification process for creating the training dataset. With the same aim, Zhou et al. [10] proposed an approach that combines text mining and data mining techniques to identify corrective bug reports in software systems, aiming to reduce misclassification noise and enhance bug prediction accuracy. Empirical studies on ten large open-source projects demonstrated its effectiveness over baseline methods and individual classifiers. Nevertheless, the approach's generalizability to commercial projects and dependence on manual training data classification still need improvement. Kallis et al. [5] proposed introducing Ticket Tagger. This GitHub app automates the issue labeling process using a machine-learning model, specifically fastText, for classifying issues such as bug reports, enhancements, or questions based on their titles and descriptions. The evaluation on a dataset of 30,000 GitHub issues demonstrated high precision and recall across categories. However, it faced challenges with false positives in questions and false negatives in enhancements, indicating room for improvement in handling diverse linguistic patterns in issue descriptions.

LLMs have also proven effective for the issue report classification problem [11, 12, 13].Nonetheless, Colavito et al. observed that the performance of these models is influenced by inconsistent and noisy labels, standard in crowd-sourced datasets [12, 14]. They proposed leveraging GPT-like Large Language Models (LLMs) for automating issue labelling in software projects, demonstrating that these models can achieve performance comparable to state-of-the-art BERT-like models without fine-tuning. However, their experiment's scope is limited, relying on a small, manually verified subset of 400 GitHub issues extracted from the well-known nlbse dataset [15], which contains more than 1.4M issues. This may affect the generalizability of the findings across more extensive and diverse datasets. Furthermore, a risk of misclassification can stem from the approach employed to deal with issues that are too long to fit within the LLM context-size limit. Indeed, the proposed approach simply truncates the reports, thus causing a loss of possible precious information.

2.3. Proposed Approach

The approach we are currently investigating for issue report classification is based on leveraging LLMs with a *dynamic* few-shot prompting strategy, with the intro-

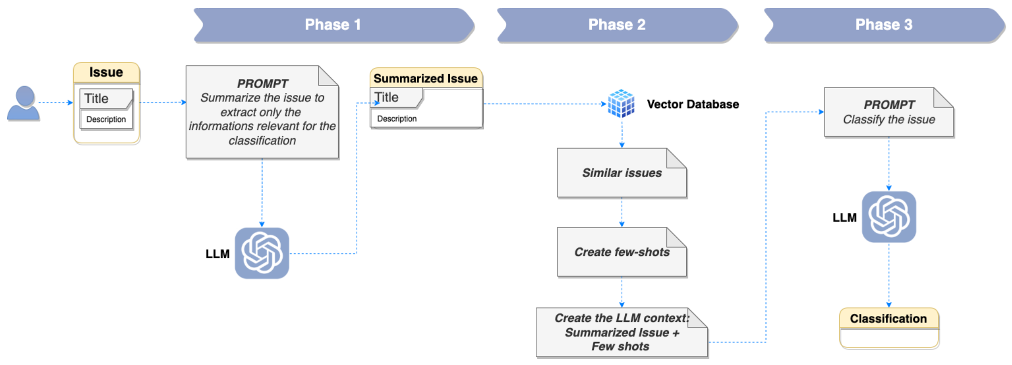

Figure 1: Issue Report Classification Process.

duction of a more advanced summarization method to manage issues that are too long to fit within the context of the LLM, and the *targeted* or *directed* selection of few-shot examples, achieved using Vector Databases. An overview of our approach is presented in Figure 1 and described as follows.

In Phase 1, we deal with issues that are too long to fit within the LLM context. In such cases, we employ the *MapReduce* programming model to summarize and parallel refine relevant data efficiently. More in detail, we partition the large issue report into smaller, manageable text chunks. Each chunk is then processed in parallel and summarized by a LLM. The result for each chunk is then combined to obtain the final, summarized report.

In Phase 2, our approach aims at selecting, as few-shot examples, issue reports that are more "relevant" w.r.t. the one that is currently being classified. To this end, we leverage a vector database such as Milvus[1]), in which previously-labelled issue reports are stored as vector representations. These vector representations are capable of capturing the semantic meaning and context of the issue reports in a high-dimensional space, and a similar vector-based representation of issues has also been used in prior works on issue report labelling [5, 7]. We then perform a similarity search between the vector representation of the current issue report to be labelled and those of previously-labelled issue reports in the vector database. This helps us identify few-shot examples that are more relevant and share common characteristics with the current issue report. Once the examples have been identified, we craft a few-shot prompt using state-of-the-art prompt engineering strategies [16], and then we present the prompt to the LLM for classification (see Phase 3 in Figure 1). We envision that providing the *right* number of relevant examples and additional context to the LLMs will further enhance their promising issue report labelling capabilities.

[1]Milvus. https://milvus.io/community

2.4. Assessment Strategy

To assess the effectiveness of our LLM-based approach for issue report classification, we propose an empirical evaluation strategy leveraging state-of-the-art LLMs such as OpenAI's GPT-4 [17], focusing on accuracy, precision, recall, and F1-score. The strategy utilizes the "nlbse 2023" dataset [15], which will be indexed into a vector database to facilitate the extraction of vector representations for selecting relevant few-shot examples for the LLM. This approach avoids fine-tuning the LLM, aiming to leverage its pre-trained capabilities to classify issue reports accurately. The assessment will compare the performance of the LLM-based method against a test set provided in the "nlbse 2023" dataset, serving as a gold standard. This comparison will focus on the metrics reported above to comprehensively evaluate the LLM's effectiveness in classifying issue reports. Classification performance will be measured using the F1-score over all four classes (micro-averaged), namely bug, feature, question, and documentation. The process involves experimenting with different numbers of few-shot examples, as well as investigating different vector representations and similarity functions to use when retrieving the few-shot examples, to identify the configuration that yields the highest performance across these metrics. By conducting this evaluation, we aim to demonstrate the potential of LLMs, like GPT-4, in automating the classification of issue reports, thereby offering a scalable and efficient alternative to manual classification methods in software development workflows.

3. LLMs for User Acceptance Test Generation

3.1. Problem Description

In software development, the generation of UATs represents a critical phase within the software testing lifecycle [18]. UATs are designed to ensure that software systems meet the specified requirements and work for the end-user as intended before the software is released. Traditionally, creating UATs involves translating user requirements and use cases into testable scenarios, requiring significant manual effort and domain expertise. This manual approach to generating UATs is time-consuming and prone to human error, potentially leading to gaps in test coverage or misinterpretation of requirements [18].

LLMs offer a promising avenue for automating the generation of UATs from natural language descriptions of software requirements or use cases. LLMs have demonstrated remarkable capabilities in understanding and generating natural language text, suggesting their potential utility in interpreting software requirements and automatically producing corresponding UATs [19, 20]. However, the application of LLMs in this context is challenging. The inherent ambiguity and variability of natural language and the complexity of software requirements pose significant obstacles to the accurate and reliable generation of UATs. Furthermore, the non-deterministic nature of LLM outputs and the limitations related to context size and model interpretability necessitate careful consideration and adaptation of these models for UAT generation [20]. The challenge lies in leveraging LLMs to convert natural language software requirements into structured UATs, requiring adapting LLMs for accurate interpretation and ensuring the UATs are comprehensive and aligned with software functionality. Overcoming these hurdles can streamline testing, boost efficiency, reduce manual effort, and improve software quality.

3.2. State of the art

Several studies have explored NLP for automating test case generation, often within specific domains or formats. Nebut et al. [21] automate system test case generation using UML and contracts, facing challenges with manual intensity and scalability in complex systems. Carvalho et al. [22] create NAT2TEST for generating test cases from Controlled Natural Language, noting reduced efficiency due to formal model reliance. Yue et al. [23] develop RTCM for converting natural language test cases into executable tests but lack comprehensive performance analysis and generalizability. Goffi et al. [24] introduce Toradocu, using Javadoc comments for test oracle generation, yet it remains a prototype with limitations in processing complex conditions. Silva et al. [25] offer a test case generation strategy using Colored Petri Nets but do not address requirement completeness and consistency, risking state explosion issues. Allala et al. [26] propose a method integrating MDE with NLP for converting user requirements into test cases, still in its initial phase and validated on a small sample. Fischbach et al. [27] explore test case automation from agile acceptance criteria, finding natural language complexity a barrier to full automation. Wang et al. [28] develop UMTG for system-level test case creation using natural language and domain models tailored for embedded systems and facing scalability challenges.

Despite the promising results, many limitations persist across the board. These limitations primarily revolve around the scalability of the approaches in complex systems, the efficiency of the processes, and the generalizability of the tools and methods to different domains or types of software systems. These limitations underscore the need for further research to integrate natural language requirements more seamlessly into the test generation process.

3.3. Proposed Approach

Our approach to automating UAT generation involves analyzing requirements expressed through use cases, specified using natural language. It consists of two primary phases: 1) Identifying the list of test cases from a use case, and 2) Elaborating the details of each test case. Throughout this process, we employ LLMs, particularly GPT-4 [17], as a tool to interpret and translate the use cases into comprehensive UAT documentation.

The initial phase tackles LLMs' context limits and non-determinism. Indeed, long textual descriptions of use cases in inputs exceeding the context limit could result in incomplete responses. At the same time, the model's non-determinism might produce inconsistent results, risking the generation of irrelevant test cases. To mitigate these challenges, we designed the prompt by leveraging the few-shot learning technique and providing precise and clear instructions for the LLM. The outcome of the identification phase is a list of test cases structured in JSON format derived from the provided text description of the use case. Each test case includes a unique identifier, a clear and concise description, the flow type, an indicator of the need for a separate UAT may not be necessary, and explicit presence in the original use case.

The second phase focuses on generating the details of the identified UATs. The goal is to produce a test case aligned with the use case scenario it refers to and sufficiently detailed to guide the test's execution without ambiguity. The details of each test case are structured in a JSON format that facilitates understanding and implementation of the tests, containing information such as preconditions, actors, and steps, including inputs and expected results. Since each test case is independent from the others, multiple requests can be processed in parallel, significantly reducing the overall execution times and optimizing efficiency and speed of execution.

To mitigate the LLM's non-determinism, we operated in multiple directions. On one hand, we focused on configuring GPT-4's hyperparameters ef-

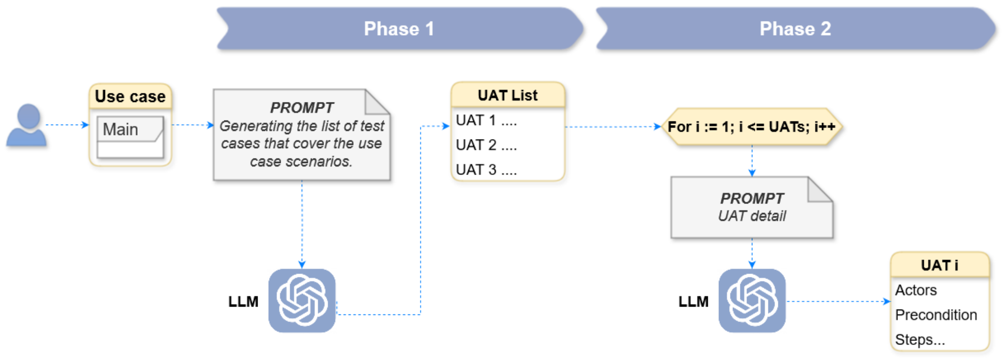

Figure 2: UAT Generation Process.

fectively. In preliminary experiments, we found that setting the `temperature`, `presence_penalty`, and `frequency_penalty` hyperparameters to 0, the `best_of` hyperparameter to 1, and the `top_p` hyperparameter to 1, as recommended by OpenAI, yielded the most deterministic outcomes.

On the other hand, to ensure GPT-4 generates specific and relevant outputs, prompts were meticulously crafted with clear, detailed instructions and examples of desired outputs, adopting a "show, do not tell" strategy [16]. This method helps the model grasp the expected format and content more accurately. Prompts and configurations underwent iterative refinements based on feedback to enhance result consistency. Finally, outputs were rigorously evaluated for consistency and requirement adherence, allowing for adjustments in response to identified non-determinism patterns.

3.4. Assessment Strategy

To evaluate the approach we will design and carry out an empirical experiment involving software engineering professionals. These participants will be divided into two groups: one utilizing our automated approach and the other resorting to manual methods for UAT generation. This design allows for a direct comparison of the outcomes, providing valuable insights into the effectiveness of the approach. By ensuring the completeness, clarity, understandability, and correctness of the generated UATs, we aim to streamline the process, enhance test coverage, and ultimately contribute to the development of higher-quality software products. Feedback from the participants will also be collected to gain insights into the usability and practicality of the approach in real-world software development scenarios. This feedback will be invaluable in refining the method and identifying areas for further research and development.

4. Conclusions

In this paper, we discuss the potential of leveraging LLMs to address two significant challenges in software engineering: issue report classification and UAT generation. By employing advanced techniques such as vector databases and few-shot learning with LLMs, we aim to enhance the efficiency and accuracy of these essential tasks. We envision that our approaches could significantly improve upon current manual and automated methods, though challenges related to natural language ambiguities and model determinism remain. Moving forward, we will focus on refining our methodologies and expanding LLM applications within software engineering to streamline development workflows and elevate software quality. Our work indicates a bright future for integrating LLMs in the field, promising substantial efficiency and product excellence advancements.

Acknowledgments

This work was partially funded by the NextGenerationEu-PNRR MUR Project FAIR (Future Artificial Intelligence Research), grant ID PE0000013.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., Gpt-4 technical report, arXiv preprint arXiv:2303.08774 (2023).
[2] H. Touvron, L. Martin, K. Stone, et al., Llama 2: Open foundation and fine-tuned chat models, 2023. `arXiv:2307.09288`.
[3] I. Ozkaya, Application of large language models to software engineering tasks: Opportunities, risks, and implications, IEEE Software 40 (2023) 4–8.

[4] G. Colavito, F. Lanubile, N. Novielli, L. Quaranta, Leveraging GPT-like LLMs to automate issue labeling (2024).

[5] R. Kallis, et al., Ticket tagger: Machine learning driven issue classification, in: Proc. of the IEEE Int. Conf. on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 406–409.

[6] O. Baysal, et al., Situational awareness: personalizing issue tracking systems, in: 2013 35th Intern. Conf. on Software Engineering (ICSE), IEEE, 2013, pp. 1185–1188.

[7] R. Kallis, et al., Predicting issue types on github, Science of Computer Programming 205 (2021) 102598.

[8] K. Herzig, et al., It's not a bug, it's a feature: how misclassification impacts bug prediction, in: 2013 35th intern. conf. on software engineering, IEEE, 2013, pp. 392–401.

[9] G. Antoniol, et al., Is it a bug or an enhancement? a text-based approach to classify change requests, in: Proc. of the 2008 Conf. of the Center for Advanced Studies on Collaborative Research, 2008, pp. 304–318.

[10] Y. Zhou, et al., Combining text mining and data mining for bug report classification, Journal of Software: Evolution and Process 28 (2016) 150–176.

[11] W. Alhindi, et al., Issue-labeler: an albert-based jira plugin for issue classification, in: 2023 IEEE/ACM 10th Intern. Conf. on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2023, pp. 40–43.

[12] G. Colavito, et al., Issue report classification using pre-trained language models, in: Proc. 1st Int. Workshop on Nat. Lang.-based Softw. Eng., 2022, pp. 29–32.

[13] M. Izadi, et al., Predicting the objective and priority of issue reports in software repositories, Empirical Software Engineering 27 (2022) 50.

[14] G. Colavito, et al., Few-shot learning for issue report classification, in: Proc. of the 2023 IEEE/ACM 2nd Int. Workshop on NLBSE, IEEE, 2023, pp. 16–19.

[15] R. Kallis, et al., The nlbse'23 tool competition, in: Proceedings of The 2nd Intern. Workshop on Natural Language-based Software Engineering (NLBSE'23), 2023.

[16] S. Ekin, Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices (2023). doi:10.36227/techrxiv.22683919.v1.

[17] OpenAI, Gpt-4 technical report, arXiv:2303.08774 (2023).

[18] B. Bruegge, A. H. Dutoit, Object–oriented software engineering. using uml, patterns, and java, Learning 5 (2009) 442.

[19] E. Kasneci, K. Sessler, et al., Chatgpt for good? on opportunities and challenges of large language models for education, Learning and Individual Differences 103 (2023) 102274.

[20] W. X. Zhao, K. Zhou, J. Li, T. Tang, et al., A survey of large language models, arXiv:2303.18223 (2023).

[21] C. Nebut, F. Fleurey, Y. Le Traon, J.-M. Jezequel, Automatic test generation: a use case driven approach, IEEE Transactions on Software Engineering 32 (2006) 140–155.

[22] G. Carvalho, et al., Nat2test tool: From natural language requirements to test cases based on csp, in: R. Calinescu, B. Rumpe (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2015, pp. 283–290.

[23] T. Yue, S. Ali, M. Zhang, Rtcm: A natural language based, automated, and practical test case generation framework, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, 2015, p. 397–408.

[24] A. Goffi, et al., Automatic generation of oracles for exceptional behaviors, in: Proceedings of the 25th Intern. Symposium on Software Testing and Analysis, ACM, 2016, p. 213–224.

[25] B. C. F. Silva, et al., Test case generation from natural language requirements using cpn simulation, in: M. Cornélio, B. Roscoe (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2016, pp. 178–193.

[26] S. C. Allala, et al., Towards transforming user requirements to test cases using mde and nlp, in: IEEE 43rd Annual Computer Software and Applications Conference, volume 2, 2019, pp. 350–355.

[27] J. Fischbach, et al., Specmate: Automated creation of test cases from acceptance criteria, in: IEEE 13th Int. Conf. on Software Testing, Validation and Verification, 2020, pp. 321–331.

[28] C. Wang, et al., Automatic generation of acceptance test cases from use case specifications: An nlp-based approach, IEEE Transactions on Software Engineering 48 (2022) 585–616.