Maintaining Requirements and Test Cases Traceability in an Agile Environment

Lucas Vieira[1], **Regina Moraes**[1], [2]

[1]Faculdade de Tecnologia (UNICAMP)
Limeira – SP – Brazil

[2]Department of Informatics Engineering (UC)
Coimbra, Portugal

`l201874@dac.unicamp.br, regina@ft.unicamp.br, remoraes@dei.uc.pt`

Abstract. *The increasing adoption of agile methodologies in software development has highlighted challenges related to the dynamic management of requirements and the continuous updating of test cases. In an environment where changes are frequent and requirements evolve rapidly, ensuring the quality and consistency of testing is essential for project success. This problem is common among various companies that adopt agile methodologies, facing difficulties in maintaining software quality while managing frequent changes in requirements. In this context, this research proposes a semi-automated approach to managing requirements and test cases by automatically identifying the impacts of requirement changes on test cases. For this purpose, the natural language process is applied with the support of the BERT language model. The methodology proposed in this study aims to enable the addition, editing, and removal of requirements and test cases in an integrated manner, ensuring that all changes are reflected efficiently and accurately. The suggested solution is based on tokenization and embeddings provided by the BERT model, combined with cosine similarity analysis, to identify the requirements and test cases that may be affected by changes in the requirement set. It is expected that the approach brings significant improvements in the efficiency of the requirement and test case management process and, consequently, in the quality of the software developed. By automatically identifying and updating test cases, it should contribute to reducing failures and rework, resulting in a more agile development process aligned with stakeholder expectations.*

1. Introduction

In today's software development landscape, the need to respond quickly to market demands and increasing competition has led companies to adopt more agile workflows. In this context, agile methodologies are widely used, offering a flexible and adaptive way to develop software. Agile methodologies, such as Scrum, prioritize continuous collaboration, constant value delivery, and the ability to respond quickly to changes. With short iterations and frequent feedback cycles, these methodologies allow teams to adjust development as customer needs evolve. This adaptability is crucial in the dynamic software market.

A characteristic of agile methodologies is the dynamic management of requirements. Unlike traditional approaches, where requirements are defined at the beginning of the project and remain static, agile methodologies (for example, Scrum) treat requirements as dynamic elements that can change throughout the project life cycle [Schwaber and Sutherland 2020]. Instead of traditionally documented requirements, agile methodologies use artifacts, such as user stories, that describe functionality from the end user's perspective and are adaptable when needed.

User story management involves the identification, documentation, analysis, prioritization, and ongoing validation of these stories. This iterative flow ensures that the product evolves in response to changing needs. Each iteration allows user stories to be reviewed and adjusted based on feedback, promoting greater alignment with customer expectations and reducing the risk of developing nonvaluable functionality. However, managing changes to requirements presents complex challenges. One of the main challenges is identifying and updating the artifacts impacted by these changes, including related test cases.

Test cases can be written using the Gherkin language [1], derived from Behavior-Driven Development (BDD), which specifies the behavior of the system through concrete examples. This language describes behavior in terms of "Given-When-Then", facilitating communication between developers, testers, and stakeholders. Each test case written in Gherkin is related to the requirements, ensuring a common understanding and validation of the requirements. These test cases are the ones that are typically manually tested by testing teams; in this context, we are not referring to code-level testing, such as Test-Driven Development (TDD), we are just taking advantage of the language. Continuous maintenance and updating of test cases are essential to ensure that all user story changes are covered. This requires a robust test management process involving reviewing existing test cases, deprecating obsolete ones, and creating new test cases to cover new requirements.

Manually managed, this process is problematic, with difficulties in identifying and updating all impacted test cases, which can result in outdated or incorrect tests that no longer reflect changing requirements. New requirements can impact current requirements, requiring a comprehensive review to ensure that all dependencies and interactions are adequately tested. A lack of effective management can lead to decreased software quality, rework, and increased costs. Without an automated approach, there is a greater risk that necessary changes will not be implemented completely or accurately, compromising the integrity and the system's functionalities. Adopting an integrated requirements and test case management process can have significant impacts on agile projects. It can improve software quality, ensure that all changes are properly verified and validated, increase development efficiency, reduce rework and costs, and promote greater stakeholder satisfaction by ensuring that the final product is always aligned with their evolving needs and expectations.

There are several approaches in the literature on generating test cases based on requirements and artifacts. [Reider et al. 2018] proposed a resource-based testing methodology using UML sequence diagrams, [Alba et al. 2022] developed a framework for GUI test scenario generation, and [Tiwari and Gupta 2015] introduced a formal methodology for TDD test case generation. However, there remains a gap in identifying which test cases are affected by changes in requirements. This poses an additional challenge for

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

development teams, who need to ensure the integrity of tests in a dynamic requirements environment. This work aims to address this challenge. The proposal resides in the automatic identification of the affected requirements based on a requirement provided as input, as well as the identification of the affected test cases related to these requirements. Using the BERT natural language processing (NLP) model [Devlin et al. 2019], the system will automatically identify impacted requirements and return the requirements and test cases most likely to be impacted and require some type of update, notifying the project manager. Moreover, the approach supports requirements in both English and Brazilian Portuguese, using BERTimbau [Souza et al. 2020] in the latter case.

In summary, as a result of the work, we will provide an approach based on NLP (in this case, BERT / BERTimbau) for the automatic identification of requirements and test cases; and will provide a graphical interface, to be used by project managers, that is capable of indicating the requirements and test cases impacted by a specific change in the requirements. This approach not only improves development efficiency, but also reduces the risks of failures and costs associated with rework in agile projects. The reduction of failures and the improvement of the requirements documentation increase the software reliability and promote greater alignment with the needs and expectations of developers and stakeholders in general.

The rest of the document is structured as follows: Section 2 presents the works more closely related to the proposed approach; Section 3 describes the approach and its foundations; a tool that gives support to the interaction between the developer and the requirements repository is presented in Section 4; Section 5 presents the results of the experiments and the tool validation process; Section 6 brings our conclusion and the indication for future work.

2. Background and Related Work

In software development, requirements bridge the gap between the needs of stakeholders and the expected software functionality. A clear and detailed understanding of these requirements is crucial to the success of any project, as they define not only what the system should do but also how it should behave under different conditions. One way to document requirements is through user stories, a widely used agile technique for capturing requirements in a user-centric manner using understandable language. In this way, they facilitate communication between developers and stakeholders. Structurally, a typical user story follows the format: ”As a [type of user], I want [action or functionality] to [benefit or value]”.

Another agile development practice is BDD, in which user stories play an even more central role. BDD extends TDD, by focusing on collaboration between developers, quality assurance (QA) managers, and non-technical stakeholders. One of the fundamental tools of BDD is the Gherkin language, which allows the creation of specifications using a simple and structured language. Gherkin makes it easy to write test scenarios that are understandable to all stakeholders, using keywords such as ”Given”, ”When”, ”Then”.

The relationship between user stories and Gherkin scenarios is close, as both focus on understanding and verifying user requirements. Each user story can be translated into one or more Gherkin scenarios, which serve as test cases to validate that the functionality meets the defined acceptance criteria. Writing test scenarios in Gherkin from user stories offers several benefits, including clarity, alignment, and the ability to automate testing. Gherkin scenarios are written in natural language, facilitating communication between all stakeholders and ensuring that everyone understands and agrees on what the system should do. In addition, these scenarios can be executed automatically, facilitating continuous verification of the system's functionality. Integrating software requirements, user stories, and test cases in Gherkin provides a robust approach to software development. According to the study by [Tsilionis et al. 2021], which compared conceptual modeling with user story mapping, the latter offers significant advantages. It not only facilitates communication between stakeholders, but also provides better adaptability to changing requirements. This study aligns with BDD practices, where well-defined user stories help create test scenarios in Gherkin, allowing continuous validation of acceptance criteria. Based on these findings, it is concluded that the use of user stories as requirements, together with test cases written in Gherkin, represents an effective approach for this work.

NLP involves several techniques and algorithms that allow the syntactic, semantic, and pragmatic analysis of texts, enabling applications such as automatic translation, chatbots, sentiment analysis, among others. NLP is based on a combination of computational linguistics, machine learning, and statistics, allowing machines to not only understand the meaning of words, but also the context in which they are used. [Sultania 2015] discussed the importance of test automation tools, highlighting early defect detection, test repeatability, and early identification of outdated or inconsistent requirements or test cases. The study by [Silva et al. 2018] investigated the impact of software specification model edits on tests generated by Model-Based Testing (MBT) in agile projects. The research revealed that 86% of the tests were discarded due to model changes, often due to syntactic improvements, affecting test maintenance and generating rework.

Another initiative for NLP is the BERT model [Devlin et al. 2019]. In the context of BERT, tokenizers are responsible for segmenting the input text into smaller units, called tokens. When BERT does not find a complete word in the vocabulary, it employs a specific tokenization method by dividing it into subunits or smaller chunks of words, allowing the model to deal with unknown words more effectively. BERT also uses special tokens, such as [CLS] and [SEP], which are inserted at the beginning of each sequence to separate different parts of the text, indicating the end of one sentence and the beginning of another. For example, ["[CLS]", "The", "user", "wants", "view", "your", "history", "of", "purchases", "[SEP]"].

Embeddings in BERT are vector representations that capture the semantic meaning of tokens. These embeddings are essential for BERT to understand the context in which words appear. The input for each token is represented by the sum of three types of embeddings: token (represents each token), segment (represents the sentence to which the token belongs), and position (represents the sequence in the sentence in which the token appears). Souza, Nogueira, and Lotufo [Souza et al. 2020] provided a version of BERT trained for the Brazilian Portuguese language so the user can also use the facilities of BERTimbau to deal with Brazilian natural language processing. In this work, both BERTimbau and BERT's tokenizers and embeddings will be used to automatically identify requirements editing and their correspondent test cases.

The methodology proposed by [Yang et al. 2019] analyzed 133 BDD projects on GitHub, focused on Java, and used the Stanford CoreNLP tool to categorize words in

.feature files and identify co-changes with 79% accuracy. Machine learning models were developed to predict when these co-changes are necessary. The results indicate that the approach helps developers identify when to modify .feature files during code commits, improving efficiency in writing tests and maintaining documentation. It highlights that an approach focused on identifying changes in requirements can significantly contribute to improving application efficiency. [Gröpler et al. 2021]'s approach presents a semi-automated approach to generate formal requirements models from natural language requirements using NLP techniques, enabling the automatic generation of test cases. The methodology involves several steps: linguistic preprocessing with spaCy (including tokenization, lemmatization, part-of-speech tagging, and dependency analysis); application of a rule-based algorithm to extract semantic content from requirements; and forming requirements models as UML sequence diagrams.

Another concern of the development team is traceability. The study by [Gazzawe et al. 2020] developed a framework to improve requirements traceability in software projects, aiming to minimize risks and facilitate the identification of relationships between artifacts, stakeholders, and the development life cycle. The results highlighted the need for traceability tools to manage changes and reduce costs and efforts, although they presented challenges in understanding their format and sequence. [Antonino et al. 2014] addresses the challenges of developing large-scale systems in agile processes, highlighting the importance of documentation and traceability between artifacts. It presents the *TraceMan* tool to manage this traceability, integrating detailed documentation with agile artifacts. The results show that comprehensive documentation, in addition to agile practices, is crucial for complex systems.

Although there are solutions on the market and in the literature that offer tools for artifact traceability, such as IBM Engineering Requirements Management DOORS (IBM, 2021), Polarion ALM (Siemens, 2020), and Jira with XRay (Atlassian, 2021), no approach was found that contemplates the automatic identification of requirements changes and the evaluation of affected test cases, a process that continues to be a significant challenge. These tools primarily focus on tracking explicit dependencies between requirements and related artifacts, including test cases, but they still rely on manual configuration or predefined relationships. Automating the identification of impacted requirements would significantly enhance the efficiency of requirements change management and provide more accurate and timely management of the impact on associated test cases.

These studies show that the use of NLP techniques can be applied to test case generation and highlight a specific gap in identifying which test cases are affected by requirements changes. Further research is needed, particularly to develop methods that provide this traceability and can efficiently identify test cases that need to be revised or updated in response to requirements changes. This work contributes to cover this gap, opening the possibility of requirements and test case analysis for future automatic identification.

3. The Approach

An essential step for the success of the approach is to establish a clear and well-defined structure for the requirements and test cases. It was decided to use user stories to define the requirements and Gherkin scenarios to develop the test cases. The user stories are usercentered, and the Gherkin language allows test cases to be written in a natural language that can be understood by all stakeholders, promoting collaboration between developers, testers, and stakeholders, ensuring everyone has a clear understanding of the acceptance criteria from the outset. Additionally, the clarity and simplicity of Gherkin test cases make the project easier to maintain and scale, contributing to long-term quality and success.

BERT stands out for its ability to consider the context of a word in both directions (left and right) simultaneously, using a transformer-based architecture. This approach significantly improves performance in several NLP tasks [Devlin et al. 2019]. This approach uses BERT's tokenizers and embeddings to automatically identify requirements editing and evaluate which test cases are affected by such changes. Furthermore, the approach supports requirements and test cases in both English and Brazilian Portuguese. In the latter case, using Bertimbau [Souza et al. 2020], a version of BERT trained for the Brazilian Portuguese language. To set the language, an input parameter should be provided to the algorithm, and then the process chooses only the set of requirements and test cases in that specific language to make the automatic selection.

In addition to BERT, it will also use cosine similarity, a measure that evaluates the similarity between two vectors based on the cosine of the angle between them. It is often used in natural language processing to compare documents and words. Mathematically, it is defined as the inner product of vectors divided by the product of the Euclidean norms of those vectors. The results range from -1 to 1. A value of 1 indicates perfectly aligned vectors, 0 indicates orthogonal vectors, and -1 indicates vectors in opposite directions.

A requirement may be associated with one or more test cases, and it may be possible to automatically identify a requirement that has been updated together with its corresponding test cases. Algorithm 1 shows the step-by-step process for this automatic selection. The algorithm receives a requirement (r), and the requirement language (l) as input parameters. Sets a requirement repository (R), and initializes an array (A) (line 1). Then, based on (l), BERT or BERTimbau embeddings model (line 2) and tokenization process (line 3) will be set on (r). Then, a loop is performed (lines 5 to 9) to obtain the embeddings of each requirement (r_i) writen in (l) in the repository (R). Finally, the vector of the requirement provided as input will be compared with the vectors of the existing (the same language) requirements in the repository (R) along with their test cases. For this, cosine similarity will be used. The closer these vectors are, the more similar the requirements will be, indicating that they are the most likely to be affected by the change in the requirements if the threshold is exceeded. This will allow us to return (line 10) the closest affected requirements and their respective test cases in the array (A). In the end, all supposedly affected requirements and their respective test cases will be available to be inspected by the developer and confirmed or modified appropriately. To facilitate the provision of input data and the visualization of results, a tool was developed, which is presented in the next section.

4. The Support Tool

To help manage the requirements in the face of essential changes, an automated system was created for requirements and test case management in agile software development environments. The proposed tool automates the identification of requirements affected by the addition of a new requirement, the alteration or removal of an existing require-

Algorithm 1: Identification of Affected Requirements

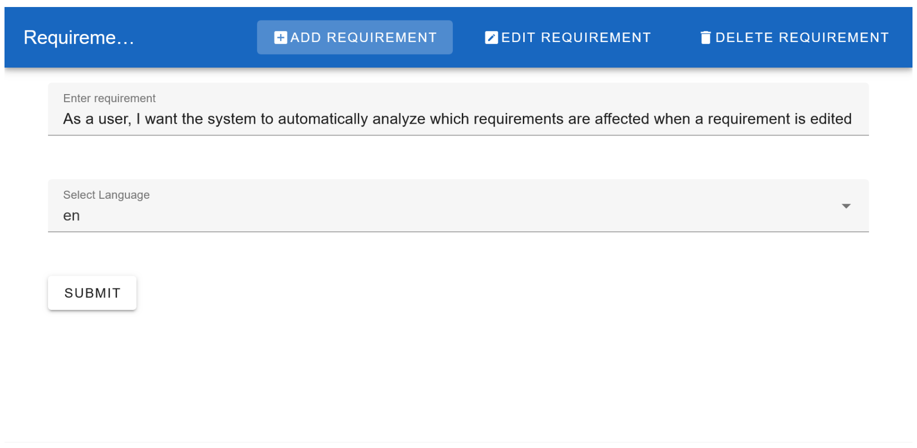
```
Data: Input requirement $r$, language $l$
Data: requirements database $R$
Result: Set of closest affected requirements and their respective test cases
1  Initialize an empty array $A$ ;
2  Assign the appropriate embeddings model based on the language $l$ ;
3  Use the tokenization process specific to $l$ to split the input requirement $r$ into
     tokens ;
4  Obtain the corresponding embeddings for each token of $r$ using the selected
     model ;
5  for each requirement $r_i \in R$ do
6      Calculate the embeddings of $r_i$ in the same way as for the input
         requirement ;
7      Compute the cosine similarity between the embeddings of $r$ and $r_i$ ;
8      if cosine similarity between $r$ and $r_i$ exceeds the threshold percentage
         then
9          Push $r_i$ into array $A$ ;
10 return $A$ containing the closest affected requirements along with their test
     cases to display to the user ;
```

ment, and the subsequent update of related test cases. The tool serves as a link between the requirements analyst and the requirements and test case repository, ensuring that any changes are automatically propagated to the affected areas, allowing the user to make informed decisions with less risk of error or omission.

The proposed software development used a set of carefully selected technologies to ensure quality, productivity, and maintainability. These included GitHub and Vue.js. GitHub is used for version control and source code storage, enabling efficient management of changes and detailed tracking of modifications. Vue.js, a JavaScript framework, powered the front-end, allowing fast creation of reusable UI components, modular development, and an ecosystem of plugins that accelerated the development process. For the back-end, Python, along with Django, were used to develop the API, enabling seamless integration between the front-end and other services. Additionally, Python's compatibility with machine learning libraries (for example, Transformers) supports the implementation of the BERT (or BERTimbau) model for the algorithm. MySQL was selected as the relational database management system for the project. It handled data storage and retrieval, offering reliability and seamless integration with Python and Django for an efficient and secure data layer.

In the process of adding a new requirement, the user creates a new requirement and sends it to the system using the tool interface (shown in Figure 1). The developer provides as an input parameter the new requirement (r) and set the language. The system uses two repositories, the requirement (R), and the test cases (T).

Algorithm 2 presents the procedure to add a new requirement. The developer elaborates a new requirement (r) to be added (line 1) and sends it to the system. The



Enter requirement
As a user, I want the system to automatically analyze which requirements are affected when a requirement is edited

Select Language
en

Figura 1. Tool interface for adding a requirement into the system

system then automatically identifies the requirements and test cases that may be affected by the addition, tokenizes them, defines the embeddings, and calculates the cosine similarity, as presented in Algorithm 1, and presents the result (line 4). The user must visually examine the conflicts and correct any inconsistencies, then confirm the addition of the new requirement in the data repository or cancel the operation. The system saves the changes and confirms to the user that the operation was successful or canceled. A new window will open to include the related test cases when the addition is confirmed. When the operations involve changes or removal of requirements beyond the steps for adding new requirements, the system retrieves the test cases that are also presented to the user to check for conflicts and necessary updates. The next section presents the experimental results and the tool validation.

5. Experimental Results and Tool Validation

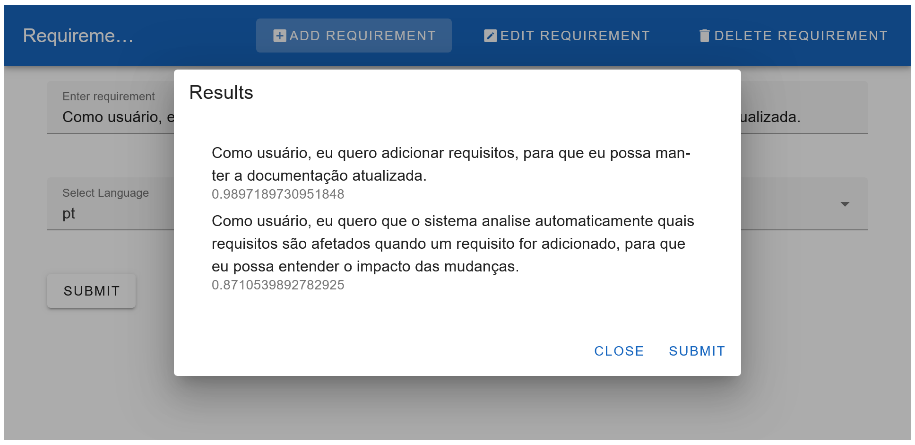
The experiments that allowed validating the proposed approach and the support tool were based on the requirements of the development of the tool itself. Therefore, the development will be done in phases to provide the opportunity to work in a development life cycle in which changes, removals, and additions occur, validating the approach. Thus, by documenting the requirements and test cases of the proposed support tool, it will be possible to verify whether the traceability between requirements and test cases is adequate. In addition, the tool will facilitate the management of requirements changes throughout the agile development life cycle, recovering the requirements and their respective test cases to avoid obsolete, outdated, and repeated artifacts. In this way, we obtained a validation of the proposed approach at the end of the development of the support tool, which is reported in this section.

In the first stage, two user stories are inserted in the repository using the tool interface so that the developer could register the initial requirements. In addition, the algorithm for automatically selecting similar requirements was created, the trigger was implemented so that this algorithm is activated with each requirement manipulation, and the interface for displaying the results was defined as a list of requirements and test cases identified by the similarity process. The test cases in this first stage also consisted of checking whether the mandatory fields had been provided by the developer in the registration form, checking whether once a requirement was added it was displayed in the list

Algorithm 2: Adding a New Requirement and Identifying Affected Requirements and Test Cases

```
   Data: Input new requirement $r$, language $l$
   Data: requirements database $R$, test cases database $T$
   Result: Set of affected requirements and test cases
 1 User elaborates a new requirement to be added ;
 2 User sends the new requirement $r$ to the system ;
 3 System executes an automatic selection algorithm to identify affected
     requirements and test cases based on the language l ;
 4 System displays the affected requirements and test cases to the user ;
 5 User analyzes the affected requirements and test cases ;
 6 User decides whether to continue with the operation ;
 7 if User decides to continue then
 8     User confirms the addition of the requirement ;
 9     System adds the new requirement $r$ to the database R;
10     System adds the new test cases $t_i$ to the database T;
11     System confirms the operation was successful ;
12 else
13     User cancels the operation ;
14     System informs that the operation was canceled ;
```

of requirements, and checking whether the trigger for the automatic selection algorithm was working appropriately. The tests were conducted in both English and Portuguese to evaluate the system's performance and adaptability across different languages. Figure 2 presents the result interface, in Portuguese language, after the insertion of the first-stage requirements. Our goal to present this interface in Portuguese is to show the versatility of the tool regarding the language.



Como usuário, e ... ualizada.

pt

Results

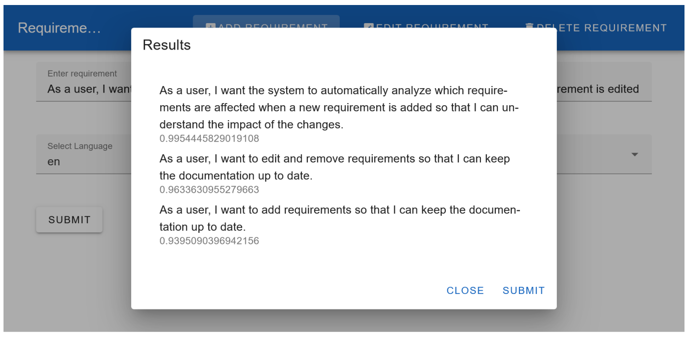
Como usuário, eu quero adicionar requisitos, para que eu possa manter a documentação atualizada.
0.9897189730951848

Como usuário, eu quero que o sistema analise automaticamente quais requisitos são afetados quando um requisito for adicionado, para que eu possa entender o impacto das mudanças.
0.8710539892782925

Figura 2. Result Interface after first-stage requirements insertion into the system

From the second stage onwards, the requirements were focused on adding new requirements, changing and removing existing requirements, as well as the relevant operations in the corresponding test cases. In the third stage, the focus was on the management of test cases, ensuring the organization and control of the system testing processes. In the fourth stage, restricted access with password security was also implemented, as well as the association of applications with the system user profile. In the last stage, a history of actions performed on the requirements was implemented. The complete list of system requirements and test scenarios is available on the following GitHub repository [2].

Figure 3 presents the results interface with three of the system requirements, showing the classification obtained according to similarity. As can be seen, all system requirements are displayed along with a probability percentage indicating the similarity likelihood, listed in descending order. This allows the user to make informed decisions regarding the requirement being added and each previous requirement in the repository. In addition, six more examples of requirement modifications were created. They served to evaluate the system's capacity to adapt to a constantly evolving requirement process. The following examples demonstrate how the system analyzed various user stories, taking into account their variations and relationships across different languages (Portuguese and English). These results highlight the system's ability to maintain consistent analysis regardless of the language.



As a user, I want the system to automatically analyze which requirements are affected when a new requirement is added so that I can understand the impact of the changes.
0.9954445829019108

As a user, I want to edit and remove requirements so that I can keep the documentation up to date.
0.9633630955279663

As a user, I want to add requirements so that I can keep the documentation up to date.
0.9395090396942156

Figura 3. Modal displayed to the user for decision-making regarding the addition of a requirement based on the probability of other affected requirements.

Example 1: Adding a Story Related to the Same Topic

Added Story (PT): "Como usuário, eu quero editar e remover requisitos, para que eu possa manter a documentação atualizada."

Result (PT): "Como usuário, eu quero adicionar requisitos, para que eu possa manter a documentação atualizada."(0.9897)

Added Story (EN): "As a user, I want to edit and remove requirements so that I can keep the documentation up-to-date."

Result (EN):"As a user, I want to add requirements so that I can keep the documentation up-to-date."(0.9841)

Analysis: The system correctly identified the high correlation between the stories of adding requirements and editing/removing requirements in both languages. The probabilities presented (0.9897 in Portuguese and 0.9841 in English) are very close, demonstrating consistency in the execution of the algorithm in both languages.

[2]https://github.com/Lucasdonivieira/managerrequirements/blob/published/requirements.txt

Example 2: Impact Analysis Between Related Features

Added Story (PT): ”Como usuário, eu quero que o sistema analise automaticamente quais requisitos são afetados quando um requisito for editado ou removido, para que eu possa entender o impacto das mudanças.”

Result (PT):”Como usuário, eu quero que o sistema analise automaticamente quais requisitos são afetados quando um requisito for adicionado...”(0.9922)

Added Story (EN): ”As a user, I want the system to automatically analyze which requirements are affected when a requirement is edited or removed so that I can understand the impact of the changes.”

Result (EN): ”As a user, I want the system to automatically analyze which requirements are affected when a new requirement is added...”(0.9954)

Analysis: The system demonstrated excellent ability to identify the relationship between stories dealing with automatic impact analysis, even when one refers to addition and the other to editing/removal. The probabilities presented (0.9922 vs 0.9954) are very high and close, indicating the consistency in the execution of the algorithm, regardless of the language.

Example 3: Addition of a Story About System Registration

Added Story:”Como usuário, eu quero me cadastrar para ter acesso ao sistema de forma segura.”

Analysis: The sorting of results was identical in both languages, with the same sequence of affected stories in both Portuguese and English. Stories related to adding requirements, editing/removing requirements, and adding/editing test cases were identified as the most impacted, maintaining the same order in both languages. It was observed that the scores in English were consistently higher than in Portuguese, but this did not affect the prioritization of the results. This consistency demonstrates the robustness of the system in identifying relationships regardless of the language used.

Example 4: Addition of a Story About Viewing Action History

Added Story: ”Como usuário, eu quero visualizar o histórico de ações realizadas com os requisitos, para acompanhar as mudanças feitas.”

Analysis: In this case, there were variations in the sorting of results between the languages. In Portuguese, stories related to automatic impact analysis were considered the most impacted, while in English, stories about adding and editing requirements received higher scores. Despite these differences in sorting, the system identified the same set of stories as being affected in both languages, maintaining consistency in the overall execution of the algorithm. This variation may indicate that the system captures specific linguistic nuances of each language.

Example 5: Editing Changing the Scope of the Add Requirements Story

Original Story (PT):”Como usuário, eu quero adicionar requisitos, para que eu possa manter a documentação atualizada.”

Suggested Edit (PT):”Como usuário, eu quero adicionar e editar requisitos, para que eu possa manter a documentação atualizada.”

Original Story (EN):"As a user, I want to add requirements so that I can keep the documentation up to date."

Suggested Edit (EN):"As a user, I want to add and edit requirements so that I can keep the documentation up to date."

Analysis: In both languages, the system correctly identified that the most impacted story would be "Como usuário, eu quero editar e remover requisitos..." with extremely high and similar probabilities (0.9957 in Portuguese and 0.9956 in English). The sorting of the results was very similar, with minor variations in the intermediate positions of the affected stories. This shows that the system is sensitive to small changes in the scope of the stories, maintaining consistency in the execution of the algorithm, regardless of the language used.

Example 6: Significant Editing Changing the Scope of the Test Case Story

Original Story (PT): Como usuário, eu quero adicionar, editar e remover casos de teste de requisitos, para garantir que minha aplicação seja testada adequadamente."

Suggested Edit (PT):"Como usuário, eu quero adicionar, editar e remover casos de teste de requisitos, e que o sistema atualize automaticamente os casos de teste quando um requisito for adicionado ou editado."

Original Story (EN):"As a user, I want to add, edit, and remove test cases for requirements so that I can ensure that my application is properly tested."

Suggested Edit (EN):"As a user, I want to add, edit, and remove test cases for requirements, and have the system automatically update the test cases when a requirement is added or edited."

Analysis: This example demonstrates how the system responds to an edit that introduces new functionality (automatic update). The system correctly identified that the most impacted stories were those related to automatic impact analysis between requirements. The probabilities were very similar in both languages (0.952/0.950 in PT and 0.959/0.955 in EN), demonstrating consistency in the execution of the algorithm, regardless of the language used. The sorting of results was also consistent between languages.

These examples provide evidence of the effectiveness of the proposed system in several key aspects. The system showed its ability to identify semantic correlations between user stories, regardless of the language used, as it maintained consistency in its analysis, even when processing different linguistic versions of the same stories; it recognizes when changes in the scope of one story affect other related stories, which is essential for impact analysis; the prioritization of affected stories was consistent across multiple languages, highlighting the system's functionality in multilingual environments. These findings support the robustness, reliability, and versatility of the automated impact analysis system for requirements and test cases, making us confident of its utility in real-world applications.

6. Conclusions and Future Work

This work presents an approach for automatic identification of requirements and test cases affected by a change in the set of requirements. The approach is based on natural language processing and can deal with both Portuguese and English written artifacts, thanks to the BERT and BERTimbau models. Moreover, the approach suggested a structured way of documenting requirements and test cases. This format helps to improve the quality and consistency of requirements and test cases writing, facilitating their traceability by the automation algorithm. To support the approach, a tool was implemented that allows manipulating the set of requirements and test cases in the repositories.

The results indicate that the approach can identify, with high precision, similar requirements when developers try to insert a new requirement. In this case, the identified requirements are presented to the user so that he can decide whether the new requirement should be included and whether other pre-existing requirements should be changed due to the new requirement. If they are changed, the test cases are also presented so that the need for their updating can also be analyzed.

In future work, the approach, as well as the support tool, will be validated in a real development environment with a larger software system. In current version, a small case study was used, and no cut-off threshold was defined or used for cosine similarity. A more in-depth study will be carried out to define the best threshold when the larger experiment is performed. Also, statistical studies will be conducted to introduce validation metrics that demonstrate the relevance and effectiveness of the approach. This will not only help verify whether the approach brings satisfactory results, but also allow for a comparison of the gains in development quality and the reduction of rework throughout the life cycle.

Acknowledgements

This work was financed by CAPES - Coord. de Aperfeiçoamento de Pessoal de Nível Superior - Brasil, finance code 001 and it was supported by the PPGT/FT - UNICAMP. Also, the authors thanks ClassApp and isaac, for the working hours exemptions.

Referências

Alba, B., Granda, M. F., and Parra, O. (2022). Ui-test: A model-based framework for visual ui testing– qualitative and quantitative evaluation. *Communications in Computer and Information Science*, 1556 CCIS:328–355.

Antonino, P. O., Keuler, T., Germann, N., and Cronauer, B. (2014). A non-invasive approach to trace architecture design, requirements specification and agile artifacts. *Proceedings of the Australian Software Engineering Conference, ASWEC*, pages 220–229.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Gazzawe, F., Lock, R., and Dawson, C. (2020). Traceability framework for requirement artefacts. *Advances in Intelligent Systems and Computing*, 1228 AISC:97–109.

Gröpler, R., Sudhi, V., García, E. J. C., and Bergmann, A. (2021). Nlp-based requirements formalization for automatic test case generation. *CEUR Workshop Proceedings*, 2951:18–30.

Reider, M., Magnus, S., and Krause, J. (2018). Feature-based testing by using model synthesis, test generation and parameterizable test prioritization. *Proceedings - 2018*

IEEE 11th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2018, pages 130–137.

Schwaber, K. and Sutherland, J. (2020). The scrum guide: The definitive guide to scrum: The rules of the game. Available online at: `https://www.scrumguides.org/`. Accessed: 18 April 2025.

Silva, A., Andrade, W. L., and Alves, E. L. (2018). A study on the impact of model evolution in mbt suites. *ACM International Conference Proceeding Series*, pages 49–56.

Souza, F., Nogueira, R., and Lotufo, R. (2020). *BERTimbau: Pretrained BERT Models for Brazilian Portuguese*, pages 403–417.

Sultania, A. K. (2015). Developing software product and test automation software using agile methodology. *Proceedings of the 2015 3rd International Conference on Computer, Communication, Control and Information Technology, C3IT 2015.*

Tiwari, S. and Gupta, A. (2015). An approach of generating test requirements for agile software development. *ACM International Conference Proceeding Series*, 18-20-February-2015:186–195.

Tsilionis, K., Maene, J., Heng, S., Wautelet, Y., and Poelmans, S. (2021). Conceptual modeling versus user story mapping: Which is the best approach to agile requirements engineering? *Lecture Notes in Business Information Processing*, 415 LNBIP:356–373.

Yang, A. Z., Costa, D. A. D., and Zou, Y. (2019). Predicting co-changes between functionality specifications and source code in behavior driven development. *IEEE International Working Conference on Mining Software Repositories*, 2019-May:534–544.