CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automated Alignment of Software Requirements and Test Cases

A Study on Developing an AI-Powered Tool for Enhancing Software Testing

Master's thesis in Computer science and engineering

SAMUEL IVARSSON
JESPER SETTERSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Automated Alignment of Software Requirements and Test Cases

A Study on Developing an AI-Powered Tool for Enhancing Software Testing

SAMUEL IVARSSON
JESPER SETTERSTRÖM

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Automated Alignment of Software Requirements and Test Cases
A Study on Developing an AI-Powered Tool for Enhancing Software Testing
SAMUEL IVARSSON
JESPER SETTERSTRÖM

Supervisor: Francisco Gomes de Oliveira Neto, Department of Computer Science and Engineering
Advisor: Daniel Nilsson, TestScouts AB
Examiner: Gregory Gay, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2023

Automated Alignment of Software Requirements and Test Cases
A Study on Developing an AI-Powered Tool for Enhancing Software Testing
SAMUEL IVARSSON
JESPER SETTERSTRÖM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software testing can be expensive, but there are numerous tools available to enhance its efficiency. Nevertheless, certain areas still lack adequate technology. To address this, we conducted a qualitative study to identify shortcomings in existing tools and determine what features are needed in new ones. Through Thematic Analysis of the interviews conducted, we discovered that system-level testing stands out as an area in need of improvement, with a particular emphasis on the importance of well-defined requirements for effective testing. In light of this, we propose the development of a tool that assists testers in aligning their tests with requirements and generates suggestions for new tests using a powerful Large Language Model (LLM). The tool demonstrated great efficacy in terms of accuracy and recall, achieving an average of 86.394% each. It also provided valuable suggestions for testing approaches related to requirements. However, we observed limitations in its ability to achieve perfect alignment between requirements and test cases, as it showed a tendency for false positives. This led to the tool having an average precision of 45.582%. Moreover, the tool exhibited high efficiency when handling smaller input sizes. However, as the input size increased, we noticed a more than linear growth in analysis time, suggesting potential scalability challenges.

Keywords: Software testing, Requirements Engineering, REST, Requirements and Test Alignment, Prompt Engineering, Large Language Models (LLM), Tool Development, Design Science, Computer Science and Engineering, Master Thesis.

Acknowledgements

We would like to express our deepest gratitude and appreciation to the following individuals and organizations for their invaluable support and contributions throughout the completion of this thesis.

First and foremost, we extend our sincere thanks to our supervisor, Francisco Gomes de Oliveira Neto, for his guidance, expertise, and unwavering support. His invaluable insights, constructive feedback, and dedication to our academic development have been instrumental in shaping the direction and quality of this thesis.

We are also grateful to our examiner, Gregory Gay, for his thorough examination of our work and his valuable feedback. His expertise and insightful comments have helped us improve the rigor and clarity of our research.

We extend our heartfelt appreciation to TestScouts, the company that collaborated with us on this thesis. We are grateful for their willingness to share their expertise and resources, which have greatly enriched our research. The insights and practical knowledge we gained from TestScouts have been integral to the development and validation of our tool.

Samuel Ivarsson & Jesper Setterström, Gothenburg, June 2023

Contents

List of Figures

List of Tables

1

Introduction

Software testing is an integral part of the software development process, as it ensures that the software is of high quality and meets the requirements of its users [1]. However, not all testing tools provide interactive features that enable testers to receive relevant feedback and support for creating and performing new tests with positive outcomes.

For instance, Lawrence et al. [2] investigated the impact of code coverage visualization on professional developers' test creation process. Their findings revealed that the tool provided a false sense of security, leading testers to believe they had completed testing when, in fact, they had not. In addition, many testing tools provide some feedback, but they are typically limited in their ability to offer insights on improving the overall test repository. This limitation can also make it difficult for testers to come up with new tests, and understand how to refactor or maintain existing tests. While reviewing existing tools on the market and the relevant literature surrounding this topic, we noticed that the tools existing to improve test suite effectiveness primarily focus on the unit level of testing, such as PMD[1], Checkstyle[2] and FindBugs [3]. The works of Lawrence et al. [2], Perscheid et al. [3], and Peruma et al. [4], also focused their research on unit-level. With this wide range of testing tools available, alongside the ongoing development of new tools, it could suggest that there is still an opportunity for enhancements at different levels of testing, such as system testing.

Moreover, as Sabaliauskaite et al. [5] explain, in software development, effective collaboration among various organizational units is critical to creating a high-quality product that is delivered on schedule and within budget. The synchronization between the requirements and verification processes is particularly important to ensure that the resulting software product meets the customers' expectations. However, the knowledge of test artifacts and their alignment with specified requirements remains a challenge [6]. In this Master's thesis, we propose a solution to address this issue through the development of a tool.

Furthermore, software testers often face challenges when joining a new project, primarily due to a lack of knowledge about the project they are joining. However,

---

[1] `https://pmd.github.io/`

[2] `https://checkstyle.org/`

[3] `http://findbugs.sourceforge.net/`

valuable knowledge is often embedded within the test artifact of a project, which is not necessarily extracted or documented by teams. With a better understanding and knowledge of test artifacts, we aim to facilitate smoother onboarding processes for new team members, minimize friction within teams, and accelerate their contribution to the testing process and product development [5], [6]. By comprehending the underlying principles and concepts of test artifacts, new testers can gain a comprehensive overview of the testing process, identify key areas of focus, and provide valuable feedback to the team. Overall, a deeper understanding of test artifacts can lead to improved quality and better outcomes for both the team and the product [7].

1.1 Problem and Purpose of the Study

Effective feedback of the System Under Test's (SUT) behavior is essential for improving the efficiency and effectiveness of software testing. According to Grant Wiggins, effective feedback should be goal-referenced, tangible, transparent, actionable, user-friendly, timely, ongoing, and consistent [8]. However, current testing tools and methods often fall short of providing the necessary interactive and ongoing feedback about the quality of test repositories. While various metrics can be used to evaluate repository quality, they do not necessarily offer the nuanced insights required for improving testing outcomes.

To address this issue, this Master's Thesis focuses on exploring which types of feedback are valuable for improving the alignment between requirements engineering and testing practices, and ultimately enhancing the overall quality of test repositories. By investigating the connections between RE and testing, we can identify and develop feedback mechanisms that are tailored to the specific needs of testers, and that provide them with clear and actionable insights into how they can improve their testing processes. Specifically, our research aims to explore the advantages of goal-referenced, transparent, timely, ongoing, and consistent feedback. The objective is to develop more efficient testing practices that can improve the quality of software.

This research aims to provide software development organizations and software testers with insight into different types of feedback that support testers in understanding the weaknesses and strengths of their test processes. We conducted an interview study to gain deeper insights and identify relevant requirements for a software testing tool.

The research emphasized the significance of utilizing a large language model (LLM) for handling diverse inputs, a common practice in system-level testing. LLMs are AI models trained on extensive text data to comprehend and generate human-like language. Additionally, the interview study highlighted the importance of requirements in the testing process. Consequently, our objective shifted towards exploring the feasibility of an automated process for extracting and evaluating quality attributes in test artifacts, specifically on aligning requirements and test cases and providing feedback on how to test specific requirements. The purpose is to determine the testing status of a requirement and suggest an appropriate testing approach.

Additionally, we investigated different prompting techniques for natural language models to address complex tasks, such as aligning requirements and test cases. Based on our findings, we developed a tool that analyzes existing tests from the software test suite and their corresponding requirements to provide information regarding whether the requirements have been tested or not. The tool's performance was evaluated in terms of various cost attributes and the accuracy of the model in analyzing test artifacts.

Moreover, this research contributes to improving the knowledge and skills of software testers regarding their test artifacts and their capabilities in designing effective tests. It also provides insight into the factors influencing the adoption and usage of feedback tools.

1.1.1 Research Questions

The research questions below define our scope, as they outline the specific areas of inquiry that will be addressed in the research.

RQ1. What are the relevant requirements of a tool that can aid testers in evaluating the quality of test artifacts?

This research question explores the characteristics of a potential software testing tool currently unavailable on the market. Our objective was to identify features that are currently lacking in the everyday tasks of professional software testers and are not offered by any existing tools, based on the available literature. To achieve this, we conducted semi-structured interviews, which were analyzed through a thematic analysis (see Chapter 3.1.1).

RQ2. What challenges and opportunities are associated with prompt engineering for assessing and aligning tests with requirements?

Based on the results from RQ1, we decided to focus on system testing, which heavily relies on natural language artifacts. Consequently, we found that leveraging language models is a viable approach for handling software artifacts expressed in natural language. This research question aims to compare and observe the outputs generated by various prompting techniques identified through the literature review. The optimization of our tool's prompts would enhance the feedback to its end-users, as well as provide further insight within the prompt engineering research domain.

RQ3. Is it possible to develop an automated process for assessing and aligning tests with requirements?

- **RQ3.1** What are the costs of performing such an analysis?
- **RQ3.2** How accurate are those analyses?

This research question and the two sub-questions center around the feasibility of creating a tool to assist testers in their testing tasks. An important aspect to consider was the cost of the tool. To address these sub-questions, a set of metrics was used to evaluate the developed tool. For instance, one metric used was the time cost of utilizing the tool, which refers to the duration between starting the tool and obtaining feedback. Additionally, a metric was used to examine the financial cost of utilizing the API used by the tool. Finally, we assessed the tool's accuracy, recall, and precision by comparing its test classification with a dataset that already had an established ground truth (i.e., a set of requirements linked to individual test cases).

1.2 Significance and Contributions of the Study

The outcomes of this Master's thesis are applicable to development practices to improve software testing. Specifically, findings can guide the design and implementation of feedback tools and methods to increase the efficiency and effectiveness of the testing process. Moreover, it can provide software development organizations and software testers with a deeper understanding of effective types of feedback and the impact of software tools on software test suites.

Furthermore, this Master's Thesis offers testers valuable insights into the importance of the alignment between requirements and tests and how LLMs can be applied to solve complex tasks such as this. The tool developed for this specific task can assist in prioritizing maintenance tasks in test code, such as determining the relevance of tests to the software's requirements, or what requirements need further implementations. It also has the potential to assist software testers in their daily activities, such as the manual labor of requirements and test alignment, or finding new perspectives on how to test a specific requirement.

1.3 Thesis Outline

This thesis outline provides readers with a high-level overview of the structure of the thesis by briefly summarizing each chapter and its contents.

Chapter 2 introduces fundamental research themes and topics relevant to this study and the succeeding chapters. The literature primarily examines various types of feedback given to software testers, their effectiveness, and their impact on the effectiveness of the test suite. It also briefly covers research related to Large Language Models and Prompt Engineering.

Chapter 3 outlines the research process, including details on data collection, analysis, and the development of the software testing tool. It comprehensively explains the steps taken to answer the research questions and justifies the chosen approach. Additionally, it explains the rationale behind the decisions made in developing the testing tool.

Chapter 4 present the analyzed data in order to answer our research questions.

The structure of this chapter is that each research question is presented as a section with the results that examine the validation of the research question.

Chapter 5 provides challenges and factors that may have influenced the results, threats to validity, and the potentials and limitations of the developed tool.

Chapter 6 serve to provide a concise summary and a closing statement on this Master's Thesis and leave the reader with a sense of resolution and a clear understanding of the main takeaways from the text.

2

Background and Related Work

Software testing is the process of evaluating a software system or its component(s) with the purpose of verifying that it satisfies specified requirements and performs its intended functions correctly [9]. Testing aims to identify defects, errors, or failures in the software and to ensure that it meets the quality standards set by stakeholders, such as end-users, customers, or management.

The testing process typically involves several stages, starting with test planning, where the testing objectives, scope, resources, and timelines are defined. Test design follows, where test cases, test scenarios, and test data are created to ensure that all functional and non-functional requirements are tested [9]. Next is the execution phase, where the actual testing takes place, and test results are recorded. The testing phase is followed by defect reporting and management, where defects are identified, reported, and tracked until they are resolved. Finally, test closure involves evaluating the testing process to identify areas for improvement and to determine if the testing objectives were met.

2.1 Different Levels of Software Testing

Software testing is abundant with numerous testing methodologies, strategies, and techniques that operate at various levels of the software [10]. The selection of which to use is largely contingent on the context and purpose of the software being tested. To simplify matters, we can broadly categorize them into two main groups: manual testing and automated testing.

Manual testing is the process of verifying and validating software applications manually, without the use of any automated tools or scripts [11]. This type of testing involves human testers who execute test cases, observe and evaluate the software's behavior, and report any defects or issues they encounter. Manual testing can be time-consuming and prone to human error, but it can also be more flexible and adaptable to changes in the software.

Automated testing is the process of verifying and validating software applications using automated tools and scripts [11]. This type of testing involves using testing frameworks and scripts to automate repetitive and time-consuming test cases, saving time and reducing the risk of human error. Automated testing can also be more reliable and repeatable than manual testing, but it can also require a significant upfront investment in testing tools and infrastructure. The manual and automated testing process involves various levels and stages, which require different individuals to perform the testing based on the level [9]. Software testing can be divided into three levels explained below: Unit Testing, Integration testing, and System testing. However, some might argue that the final step of any software testing process is Acceptance Testing [12], which is why it's included here as well.

Unit Testing: Testing on the unit level mainly tests individual units or components of the software application [9]. The objective is to verify if each unit functions correctly and per the specified requirements in isolation. Meaning that when testing these separate functions, they should not communicate with other modules or components. Developers typically carry out unit testing using frameworks such as JUnit and NUnit.

Integration Testing: This type of testing involves combining different units or components of the software and testing them as a group to ensure they work correctly together [9]. Integration testing may involve testing multiple levels of the application stack and can be performed either manually or using automated testing tools.

System Testing: It is a comprehensive testing process of the entire software system to verify that it meets the specified requirements [9]. Meaning that the software, with all its different components and modules, is integrated to verify whether the software works as expected or not. System testing is typically carried out by a dedicated team of testers who are not involved in the development of the software. The objective of system testing is to identify any defects, bugs, or functional gaps in the software application.

Acceptance Testing: This type of testing involves verifying if the software application meets the expectations and requirements of the end users or stakeholders [12]. Because even though the system testing might be complete and all requirements were met, the finished software might not be what the end users expected it to be. Acceptance testing can be performed either manually or using automated testing tools. It is the final testing stage and is typically performed before the software application is released to production.

Overall, the different levels of software testing are aimed at ensuring that the software application meets the requirements and expectations of the stakeholders, and is free from any defects or bugs that may adversely affect its performance or functionality.

2.2 Large Language Models

Relatively new breakthroughs have been shown in the field of Artificial Intelligence, especially by utilizing Large Language Models (LLMs). Trending tools such as

ChatGPT[1], BERT[2], and GitHub Copilot[3] employ these LLMs to generate data, such as text and code. The potential use cases of such tools have been widely debated in various areas, such as education, climate projections, public health, finance research, and medical writing, as highlighted in the references [13]–[17].

Although the previously mentioned areas are just a few instances and could be expanded, these tools have limitations like any new technology. To begin with, they generate data that is based on the datasets they were trained on. Consequently, the outputs may incorporate any biases present in the training data, which could lead to misleading or erroneous results that deviate from factual truth. Furthermore, the outputs may lack semantic coherence and appear nonsensical or gibberish [18]. With these limitations in mind, we will investigate whether we could extend the list of use cases by adopting LLM tools to fulfill the requirements for our developed tool, see Section 4.1.

Typically, these tools require plain natural language text as input, which often includes instructions on how the model should respond. Such inputs are commonly known as prompts. For instance, a user may design regulations, automate processes, and define particular attributes for the LLM to attain a desired outcome [19].

When creating prompts as a systematic practice to generate a desired output, it can be referred to as prompt engineering. Dang et al. [20] highlights opportunities and challenges when using prompting techniques such as zero-shot, one-shot, and few-shot prompting. One particular opportunity they mention is "End-User Programming of Creative Tools", where they emphasize that tools can utilize conversations between humans and the model to complete specific tasks. Moreover, studies have shown insights into how to design prompts more effectively. For example, Lu et al. [21], showed that the ordering of words matters. Beltagy et al. [22], found a solution on how to process longer inputs with great empirical results by fine-tuning a transformer. However, Wu et al. [23], argue that the LLMs can be less effective for more complex tasks, where the prompts tend to grow larger. Nevertheless, they introduced the concept of "Chaining LLM steps together", improving the outputs' quality.

2.2.1 Prompting techniques

In this section, we elaborate on several prompting techniques used when we applied prompt engineering for our tool. By utilizing the mentioned techniques on a specific use case, our Master's thesis aims to extend the domain of prompt engineering and achieve the best possible outcome for our tool.

Zero-, One-, and Few-shot prompting: As Dang et al. [20] mention, examples in the text prompt determine the categorization of prompting as zero-shot, one-shot, or few-shot. Zero-shot prompting refers to prompts without specific examples, like

---

[1] `https://chat.openai.com/`

[2] `https://cloud.google.com/ai-platform/training/docs/algorithms/bert-start`

[3] `https://github.com/features/copilot`

“Derive X in the following formula: 1+1=X”. On the other hand, one-shot learning involves providing a single example, while few-shot learning involves providing multiple examples within the prompt. Including one or more examples in the prompt has the potential to enhance the quality of the output.

Instruction with prompt: Ni and Li [24] define this technique as an iteration of Few-shot prompting. This is by providing multiple examples of how the generated output should look, while also providing more instructions and rules for the LLM. This process aims to guide the LLM in generating more accurate and contextually appropriate responses based on the given prompts, by combining input texts and handwritten prompts.

Role-playing: Li et al. [25] established a framework in which a user inputs specific roles that the LLM should immerse itself in, namely “Role-Playing”. The key concept is that the LLM should address specific tasks based on the given role’s perspective. For instance, if specifying an LLM to immerse itself as a toddler with poor grammatical skills while also providing a task, the generated response will likely contain nonsense related to a toddler’s daily life. By giving the LLM more sophisticated roles, the generated outputs could provide new insights and perspectives that the creator of the prompt may lack.

Chaining LLM steps together: Wu et al. [23] found that when LLMs were given more complex tasks their responses were less effective. To overcome this challenge, they introduced the concept of Chaining LLM steps together. This allows the generated output from one prompt to serve as input for the next prompt, enabling an iterative process that enhances the quality and coherence of the LLM’s responses.

Chain-of-Thought Prompting: Wei et al. [26], introduce the technique called Chain-of-Thought Prompting. While it shares some similarities with the Chaining LLM steps approach, this technique emphasizes generating a chain of thought consisting of intermediate reasoning steps. Although it still utilizes the concept of chaining, the key difference lies in reducing the complexity of each prompt. By simplifying the prompts and giving suggestions on how to reason, the technique can yield remarkable empirical improvements in the performance of language models.

2.3 Related Work

In previous studies, feedback has been investigated through a variety of methods, such as metrics and quality measures [27], surveys or interviews with testers [3] or code coverage criteria [1]. These studies have provided valuable insights into the importance of feedback for improving the efficiency and effectiveness of software testing and what type of feedback might be useful.

Hemmati discusses the effectiveness of code coverage criteria as a metric to measure the adequacy of test cases/suites in [1]. His study evaluated several existing and standard control and data flow coverage criteria on a set of developer-written fault-revealing test cases from several releases of five open-source projects. The main finding of his study is that basic coverage criteria, such as statement coverage, are weak in detecting faults, and combining several control-flow coverages together can detect more faults than using the strongest criterion alone. The study also found that a basic data-flow coverage can detect many undetected faults, and on average, 15% of the faults may not be detected by any of the standard control and data-flow coverage criteria. In other words, he found that code coverage criteria can identify areas of code that need more testing, but it does not necessarily describe the quality of the tests. This means that the proposed feedback tool should not base its analysis solely on the traditional coverage criteria as many other tools do, since it will not be sufficient.

Perscheid et al. [3] reported that providing feedback to testers during the testing process can assist developers in identifying missing tests and correcting low-quality tests. This was achieved by building a tool that can track areas within the system that need refinement and identifying an expert for the targeted area. Meaning that the tool suggests which developer is most fit to update the inadequately tested area. The suggestion was based on previous commits within the repository and suggests the developer with the best insight for the artifact. This study is similar to ours, but there is a key difference, their study focuses on improving certain areas of a test suite, while ours has a greater focus on assisting the tester in writing better tests, in terms of quality. Quality could refer to tests that are better at identifying faults, are easy to read and understand, and are easy to maintain. However, the mentioned quality traits mainly focus on the actual code quality and do not always specify how good or effective a test suite actually is. For example, findings such as the empirical evidence presented by Rempel and Mäder [28].

Rempel and Mäder establish that increased traceability reduces the anticipated defect rate in the resulting software, which also marks the first instance of such findings. The study analyzed the relationship between traceability completeness and defect rate across different software development processes. The authors collected data on the completeness of traceability links between requirements, design, and code artifacts. The data were analyzed in a multi-level Poisson regression analysis to determine the extent to which traceability completeness could explain variation in the defect rate, after controlling for other variables such as project size and development methodology. The results of the study showed a strong correlation between traceability completeness and defect rate, suggesting that improving traceability can lead to significant improvements in software quality. This implies that traceability holds great practical significance for all kinds of software development projects. Although traceability is not a mandatory requirement under a regulation or standard, it can be advantageous for software testers when implementing or improving test suites in order to ensure correctness in the developed software.

In contrast, studies such as Roohullah et al. [10] and Basili and Selby [29] have shown that the effectiveness of software testing can vary depending on the types of testing techniques used by testers of different skill levels. This should be taken into consideration when analyzing the results of the case study. However, none of the papers above specifically points out the most useful feedback type.

In addition, studies have investigated the use of code coverage visualizations by professional developers [2]. This paper's findings suggest that visualizing code coverage can assist testers in writing tests. As pointed out by Hemmati, it's important to note that code coverage alone is not a definitive measure of test quality. Nevertheless, the visualization aspect can serve as a useful inspiration for presenting feedback from our tool.

Zhang and Mesbah [30] discuss the correlation between the use of assertions in test cases and the test suite's quality. They conducted an empirical study that aimed to investigate the correlation between the quality of a test suite and various factors such as the number of assertions, the extent to which assertions are used, and the different types of assertions used. The study found that test suites with a high number of assertions tend to be more efficient in detecting defects. The results of the study indicate that assertion coverage may be a useful method for providing feedback with our tool, as it has been demonstrated to enhance the ability of the test suite to detect faults.

Enoiu et al. [31] did a study that is about understanding the cognitive processes of software testers during the testing process. The study presents a cognitive model of software testing based on how problem-solving is conceptualized in cognitive psychology. The study proposes an experiment that uses verbal protocol analysis to understand the mechanisms by which human testers choose, design, implement, and evaluate test cases. An initial evaluation was performed with five software engineering master students as subjects. The main findings of the study are that the results support a problem-solving-based model of test design for capturing testers' cognitive processes. Though the study only involved student-level software testers and needs more study to confirm the results, it provides valuable information on when feedback is most useful to be received, since the model that was found can be divided into steps.

Enoiu et al. did another study that aimed to find out the level of awareness of cognitive biases among software testers [32]. A survey was conducted with 60 practitioners from different environments. The results indicate that testers are aware of biases such as confirmation bias, fixation, and convenience. This is an important factor to take into account in our research, as the participants of our study will also be software testers.

In addition, this Master's Thesis will also contribute to the larger, ongoing dialogue in the literature about the topic by filling in gaps regarding the impact of different types of feedback and extending prior studies such as [3], [27]. By providing a benchmark for comparing the results with other findings and elaborating on the literature review, this Master's Thesis aims to refine, revise and extend the current knowledge of this field.

3

Methodology

This methodology section provides an overview of the research approach and methods used in this Master's Thesis. It includes details on the execution and planning of the thesis, the data collection procedures, and the methods used for data analysis.

The Master's Thesis was conducted with TestScouts [1], which is a software testing company for consultants in testing, with 23 employees. They specialize in providing quality assurance and testing services for almost all types of software tests. They provide services such as test automation, development of test environments, development of test architecture, and more. As the thesis is in collaboration with TestScouts, the subjects used for the different interviews and surveys will solely be employees within the company. The subjects in question have different areas of expertise, experience, and knowledge. However, they still have their main focus on software testing, which makes them suitable subjects. The diversity that emerges within the range of subjects could enhance the thesis's robustness, as the subjects will contribute to a wider perspective.

The research methodology employed in this paper is based on the principles of design science [33]. While action research concentrates on evaluating the outcomes of a particular action in a case study, design science prioritizes the actual process of designing and building a design artifact [34]. This means that when implementing a solution in a case study, design science focuses on the creation of the design artifact itself, rather than solely on its effects. The design science method aligns well with the aim of this paper, which is to create a software tool. The tool itself will be the design artifact. The method will also help us evaluate our tool in the context of a company.

3.1 The Design Science Method

The thesis was be divided into three cycles, roughly one month each, where most of the focus within the cycle will be put on the different research questions. We will expand our literature review and evaluate and refine the prototype within each cycle. This is illustrated in Figure 3.1. Each cycle and its activities will be further elaborated in the sections below.

---

[1] `https://testscouts.se/`

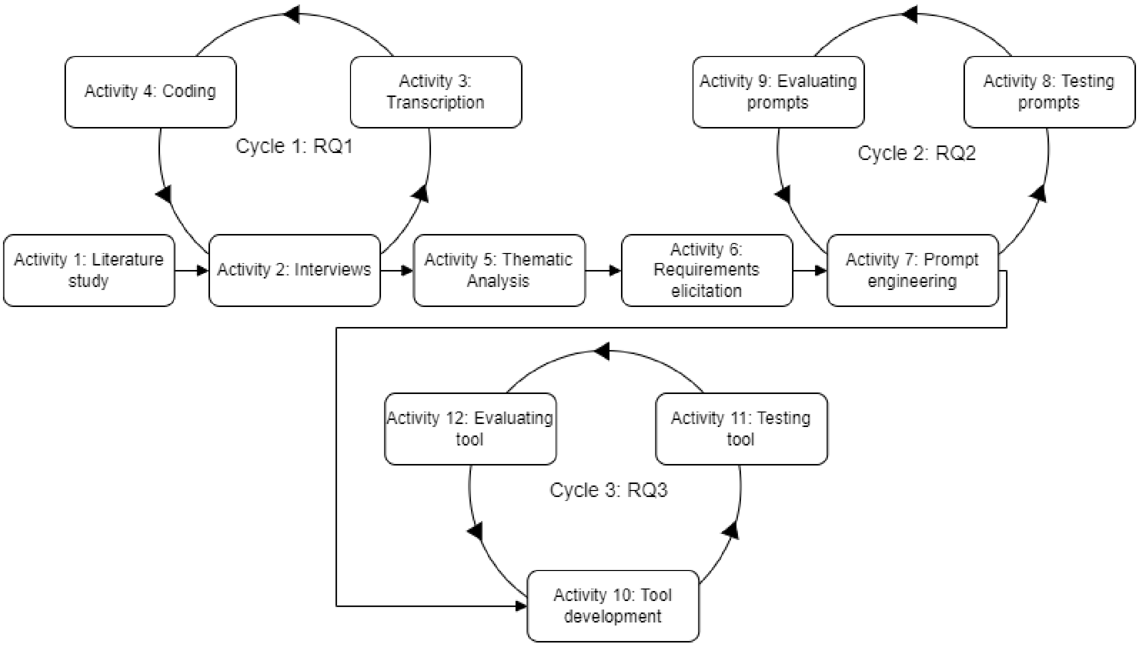


Figure 3.1: Schema of the design science cycles.

Our goal was to develop a tool that provides feedback based on analysis of existing tests of the software test suite in a realistic context. In design science, the development of an artifact involves gathering requirements and identifying features. The delivery timing, type, and frequency of feedback for the tool are determined based on the results observed in each cycle; see Chapter 4. Test artifacts and feedback differ depending on the test levels, such as unit testing, integration testing, or system testing, and which we will focus on was decided based on the results from the elicitation interviews conducted for the first cycle of the thesis.

3.1.1 Cycle 1: Exploring the Significance of Test Feedback

During the initial cycle of the process, the primary objective was to construct a comprehensive understanding of the current market offerings, effective types of feedback supported by research, and common challenges faced by professional software testers. These various aspects gave us the insight needed to develop the requirements specification for our tool. Figure 3.2 presents the steps we took to achieve our deliverables for Cycle 1.

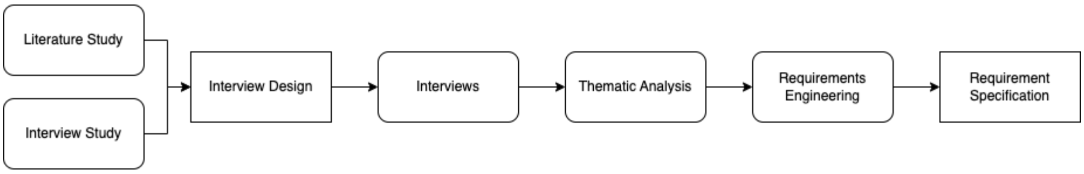


Figure 3.2: Overview of the steps we took to achieve our deliverables for Cycle 1. The rounded rectangles are activities, and the sharp-edged rectangles are deliverables.

Literature Study: The literature study aimed to investigate two main areas related to software testing: types of feedback supported by research and the effectiveness of different types of feedback. This could include feedback related to defects, test coverage, or other aspects of the testing process. By understanding what types of feedback have been shown to be effective, the study aimed to inform the development of a tool that would provide meaningful feedback to software testers.

In turn, literature on the effectiveness of different feedback could include comparative studies of different feedback methods or examinations of how specific feedback techniques impact software testing outcomes. By understanding which feedback approaches are most effective, the study aimed to ensure that the tool being developed would provide high-quality feedback to users. By leveraging this knowledge, the study aimed to inform the development of a tool that would offer effective and valuable feedback to software testers.

Interview Study: With the insight from the literature study, we began an interview study, in which we prepared a set of semi-structured interviews. The interviews aimed to explore participants' perceptions and experiences related to software testing to extract requirements for our tool. The interview guide, shown in Appendix A, consisted of open-ended questions and prompts designed to elicit detailed and nuanced responses from participants. The interview questions were developed based on a review of the relevant literature and in consultation with our research supervisor.

The participants for the interviews were employees at TestScouts. To retain the integrity of the participants, the data collected was anonymized. The participants had varying levels of experience as testers, and they held diverse positions at TestScouts. An overview of the participants' experience level is presented in Table 3.1. As only three interviews were held, we cannot claim that these requirements are the absolutely most needed for a tool that currently does not exist on the market. Instead, the goal of the study had a greater focus on eliciting relevant requirements for the tool, rather than gaining a deep understanding of the current practices done at the company. However, with these interviews, we did reach saturation.

Table 3.1: Table over participants with corresponding participant ID, Role, and Experience in years.

ID	Role	Experience in years
P1	IT manager, Software Tester	24
P2	Software Tester	21
P3	Test Engineer	6

We conducted face-to-face semi-structured interviews with three participants at TestScout's office, which lasted approximately 45 minutes on average. The interviewees were identified through discussion with our advisor, Daniel Nilsson, who deemed them suitable. As both the interviewees and we as interviewers' native language is Swedish, the interviews were conducted in Swedish. All three participants consented to a contract, which stated that the participants would remain anonymous, the audio of the interviews were recorded and participants were free to opt out of the study at any time. Appendix A presents the interview guide, which consists of three parts. In the first part, we asked participants about their role in the company and their length of experience in the field. In the second part, we asked participants about their process when starting a new project, including their initial steps, what they look for when trying to improve existing work, and how they validate their work. Lastly, we posed tool-specific questions to the participants, such as their desired feedback in daily workflow, preferred tool environment, and desired features for a tool of this nature.

Overall, the interview instrument was designed to be semi-structured, allowing for some flexibility and adaptability in the interview process while maintaining a clear focus on the research question and themes of interest.

Interview Analysis: We analyzed the interviews, which were conducted in Swedish, through a thematic analysis in an inductive manner [35]. To facilitate labeling and theme creation, we transcribed the interviews in Swedish and subsequently translated them into English when presenting them as part of our results (Chapter 4). Our rationale for using English was that it offers a richer vocabulary for discussing software testing and simplifies the presentation of our findings in this paper. In order to maintain consistency, we independently coded each transcript by highlighting what we considered to be significant aspects of the participant's responses. When comparing our codes, we achieved a high level of agreement. Our analysis revealed that the coding of the three transcribed interviews independently resulted in a 74.2% similarity of codes. This was measured by dividing the number of codes that were identical with the total amount of codes extracted. We entered these codes onto a Miro board as post-it notes and grouped together those codes that conveyed similar ideas. Labeling is the process of extracting the terms used to categorize segments of information that accurately represent the fundamental nature of the data [36]. By labeling these groups, we were able to identify themes that encompassed multiple groups.

3.1.2 Cycle 2: Tool Development

Within this section, we'll describe how the tool evolved and how the requirements identified in Cycle 1 were fulfilled. The requirements can be found in Section 4.1. Figure 3.3 presents the steps we took to achieve our deliverables for Cycle 2.

A user interface was created to establish the tool's usage and how data should be presented. TeleportHQ[2] was used to design the interface, which is a platform that promotes collaboration in front-end development by offering integrated tools for UI development and content modeling. Their visual builder is a robust tool that allows users to produce and publish headless static websites. To create a standalone web-based application, the decision was made to use React.js[3], a popular

---

[2] `https://teleporthq.io/`

[3] `https://react.dev/`

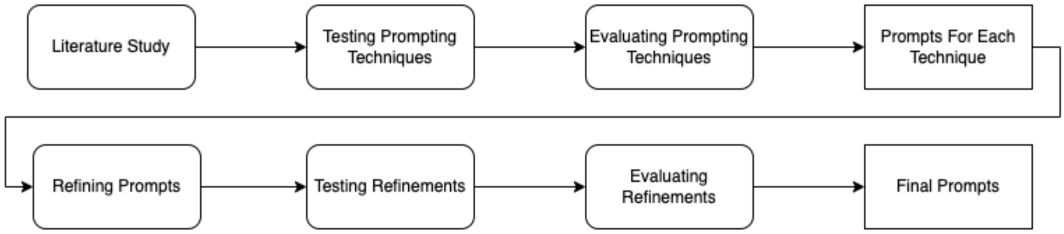


Figure 3.3: Overview of the steps we took to achieve our deliverables for cycle 2. The rounded rectangles are activities, and the sharp-edged rectangles are deliverables.

JavaScript library for building web applications. Additionally, React is created and maintained by Meta Platforms, Inc[4] (formerly Facebook Inc), a well-known tech company, which provides confidence that the library will continue to receive support and enhancements in the future.

Given that our tool was intended to incorporate an LLM, we needed to explore which model to adopt and how to integrate it. As the AI model market has flourished in recent times, a multitude of opportunities have emerged. While taking into account feedback from supervisors, colleagues, and various online reviews, we made the decision to utilize the model GPT-3.5-turbo[5], more commonly known as GPT-3.5. This model, developed by OpenAI[6], currently stands as one of the largest models in the market, boasting 175 billion parameters [37], which made it a natural choice. To employ the model, we communicate with the OpenAI API, which involves sending an API request with a string-formatted message. Subsequently, the model generates a response, which facilitates a chat-like interaction between us, the developers, and the model.

Different prompts were tested in order to receive a deterministic response, which should fulfill the requirements for the tool. More specifically, the feedback generated by the prompts should include information on whether a test suite is testing its relevant requirements, and it should also provide the connection between a requirement and its corresponding test case(s). Since the LLM generates responses based on the data its been trained on, the responses may vary, even though the input is the same. To decide on how to prompt the model to generate the desired response properly, we followed various techniques shown in the literature review, see Section 2.2.1. This was done in a systematic trial-and-error procedure, where we followed the aforementioned prompting techniques. The prompts were iterated until they worked for some sample data. We then extended or altered the sample data and noticed deviations in the response, in which we altered the prompts. By repeating these steps, we found a solution that worked as intended with our sample data. See Section 4.2 for a more detailed explanation.

The data that was used for testing the prompting techniques was a requirements

---

[4] `https://about.meta.com/`

[5] `https://platform.openai.com/docs/model-index-for-researchers`

[6] `https://openai.com/blog/openai-api`

specification and its corresponding test cases for a Bluetooth[7] profile, namely the Headset Profile 1.2[8]. The profile included several documents, namely the Headset Profile 1.2, which outlined the requirements, the Test Suite (TS) containing the test cases, and the Implementation Conformance Statement (ICS) that provided the alignment between the requirements and test cases. The profile is an established standard of how to certificate a headset product with Bluetooth. Combining this with the fact that the mapping and alignment between requirements and test cases were extensive, it was deemed realistic and practical as test data. The data extracted from the documents contained 10 requirements and 10 test cases. One requirement lacked a specified test case, and one test case had no corresponding requirement. The remaining requirements and test cases were in a 1:1 ratio alignment.

The documentation provided by Bluetooth was delivered as PDF (Portable Document Format) files. Through discussion with colleagues and supervisor, we had formerly decided to optimize the tool for CSV (Comma-Separated Values) files. This is because software testers work with numerous different tools for requirement and test suite documentation. However, the majority of these tools can opt to export the contents of the tool to CSV format. Because of this, the data provided by Bluetooth was manually exported to CSV format in order to fit the tool. This could prove to be contradicting the requirement that specifies that the tool should be capable of processing various inputs. However, as the requirement only specifies the contents of the input and due to time limitations, it was decided only to support CSV formats as inputs.

With the aforementioned dataset, we found prompts that repeatedly succeeded in mapping the correct test cases to their corresponding requirement. The different prompting techniques and different sets of prompts are more thoroughly described in Section 4.2. With these prompts finalized and the generated response presented in a readable and user-friendly way, the remaining requirement was to provide the tester with suggestions on how to test said requirement. By implementing this, the tool was deemed complete.

3.1.3 Cycle 3: The Tool and its Evaluation

This section provides a summary of the tool's functionalities and architecture. We'll also briefly discuss how we proceeded to evaluate the tool in terms of cost and accuracy. Figure 3.4 presents the steps we took to achieve our deliverables for Cycle 3.

The developed software testing tool is a web-based application that can assist software testers in their daily activities by leveraging artificial intelligence through automation. The tool's front-end or home page, allows users to input two files: (i) one for their software requirement specification, and (ii) another file with the corresponding test suite. Both these files should be in CSV formats as this is what's currently supported by the tool. This interface is visualized in Figure 3.5.

---

[7] `https://www.bluetooth.com/`

[8] `https://www.bluetooth.com/specifications/specs/headset-profile-1-2/`

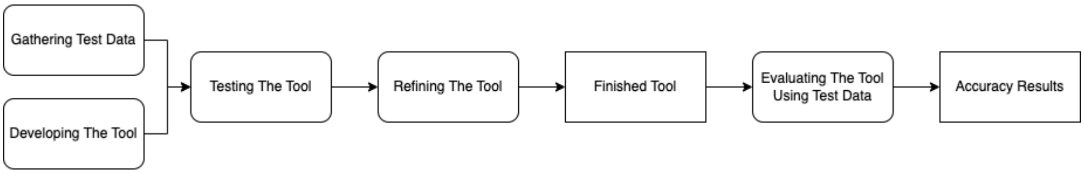


Figure 3.4: Overview of the steps we took to achieve our deliverables for Cycle 3. The rounded rectangles are activities, and the sharp-edged rectangles are deliverables.

By pressing the button “Run Analysis”, the two files are parsed and several prompts are made to the LLM through API requests. The number of requests depends on the number of requirements and test cases in the files. Once the API responses are received, they are processed and presented in a readable format for the end-user to comprehend. To see more concretely how the result is presented, see Figure 3.6.

As Figure 3.6 depicts, a table is used to display the results, where each row represents a requirement and its corresponding result. Additionally, the table includes a header with five columns labeled as *Req. ID, Description, Tested, Tested by*, and *Do you want testing suggestions for the requirement?*. The Req. ID is derived from the input file and serves as the name or ID for each requirement. The Description column provides a description of each requirement, also extracted from the input file. The Tested by column displays a list of all the test cases that have been used to test the requirement. These test cases are represented by their ID or name combined with a description of the test case, which also is extracted from the input file. Lastly, the column titled “Do you want testing suggestions for the requirement?” features a “Yes” button that, when clicked, sends an additional API request to the LLM. The response the request provides is a suggestion on how to test the requirement, which then replaces the “Yes” button. To run a new analysis, the user can click the “New Analysis” button, which is always displayed in the top left corner of the page.

Figure 3.7 depicts the architecture of the software testing tool. It shows the components of the tool, including the front-end, back-end, the API, and the LLM, as well as the flow of data between these components. The figure provides a visual representation of how the tool works and helps the reader understand the tool’s functionality.

The evaluation process of the tool involved two main aspects. The first aspect was to determine the different costs associated with the tool, which included the financial cost of using the OpenAI API and the time cost of utilizing the tool. The second aspect was to evaluate the tool’s accuracy in aligning requirements and test cases. A detailed elaboration on this can be found in Section 4.3.

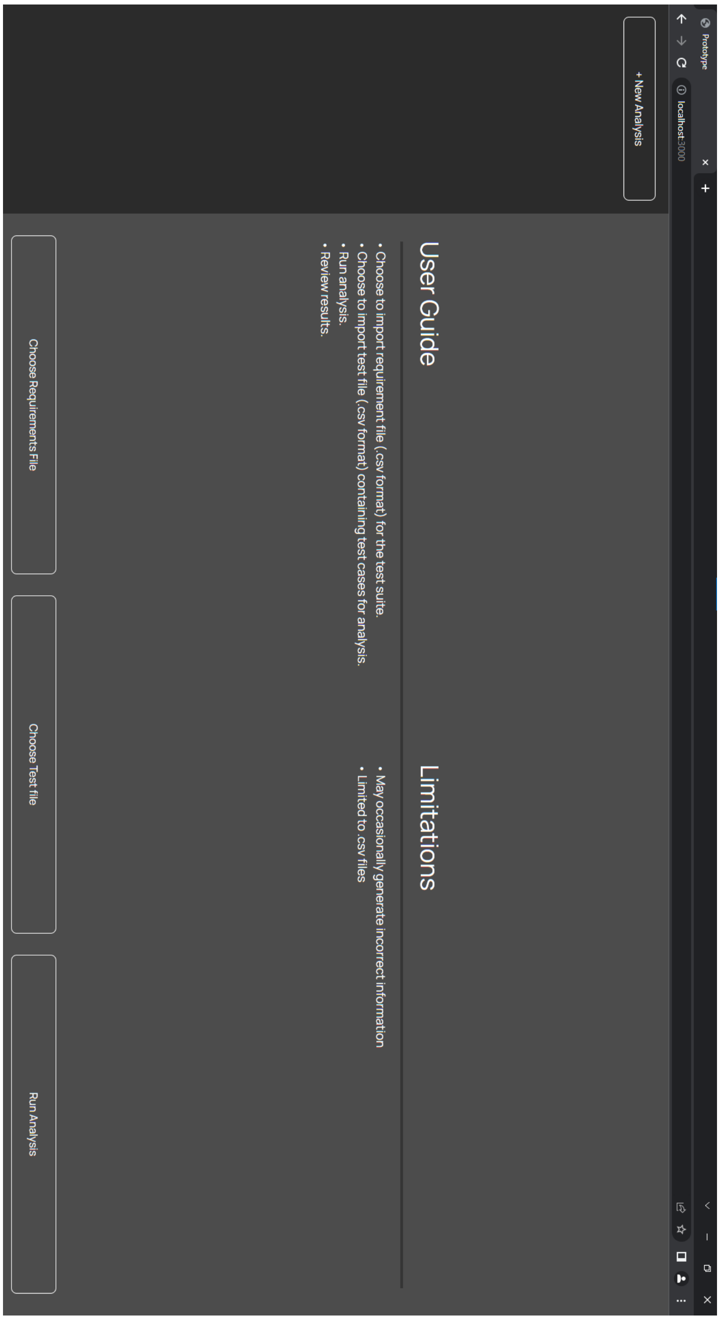
A more extensive dataset would be necessary to ensure the results’ reliability and validity. We utilized a dataset obtained from Mozilla’s front-end quality assurance archives of test cases for the evaluation. Specifically, we collected a set of 49 requirements and 44 test cases from the *Bookmark*[9] and *History*[10] features. This assisted our research greatly since a ground truth of the alignment between requirements and test cases had already been established. Due to the associated costs of running an analysis with the tool, we only performed three runs on the Mozilla dataset. From each run, we collected the number of True Positives (TPs), False Positives (FPs), True Negatives (TNs), and False Negatives (FNs) in order to calculate the accuracy, recall, and precision of the tool. We did this by analyzing the results from the tool's output page and entering them into a spreadsheet. See Figure 3.6. We also collected the usage time and analysis time from each run by utilizing the built-in javascript function "console.time()". We also calculated the total cost of running an analysis with any number of requirements and test cases. We did this by calculating the cost for the different sections of our prompts by using OpenAI's tool called Tokenizer [11]. For a more thorough and detailed explanation of the results and metrics of RQ3 and its sub-questions, see Section 4.3.

---

[9] `https://www-archive.mozilla.org/quality/browser/front-end/testcases/bookmarks/`

[10] `https://www-archive.mozilla.org/quality/browser/front-end/testcases/bookmarks/`

[11] `https://platform.openai.com/tokenizer`



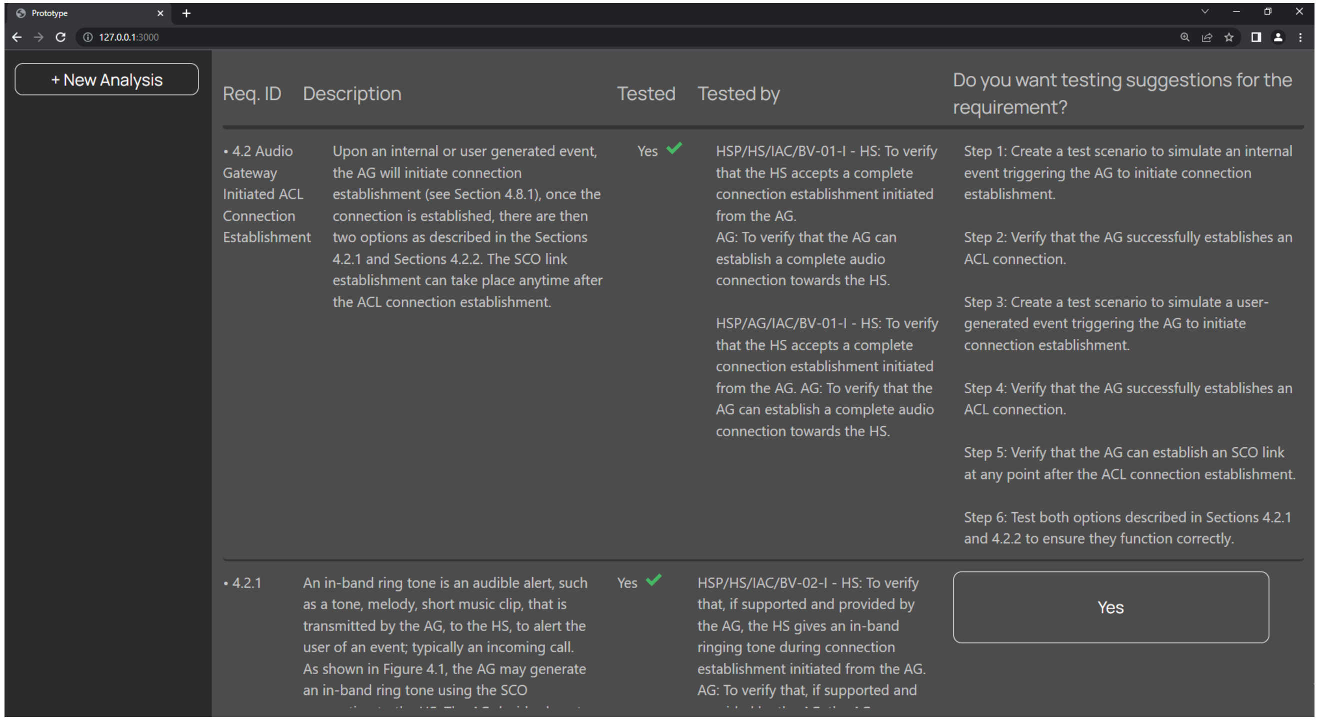
User Guide

- Choose to import requirement file (.csv format) for the test suite.
- Choose to import test file (.csv format) containing test cases for analysis.
- Run analysis.
- Review results.

Limitations

- May occasionally generate incorrect information
- Limited to .csv files

Figure 3.5: The initial page or home page of the tool.



Req. ID	Description	Tested	Tested by	Do you want testing suggestions for the requirement?
• 4.2 Audio Gateway Initiated ACL Connection Establishment	Upon an internal or user generated event, the AG will initiate connection establishment (see Section 4.8.1), once the connection is established, there are then two options as described in the Sections 4.2.1 and Sections 4.2.2. The SCO link establishment can take place anytime after the ACL connection establishment.	Yes ✓	HSP/HS/IAC/BV-01-I - HS: To verify that the HS accepts a complete connection establishment initiated from the AG. AG: To verify that the AG can establish a complete audio connection towards the HS. HSP/AG/IAC/BV-01-I - HS: To verify that the HS accepts a complete connection establishment initiated from the AG. AG: To verify that the AG can establish a complete audio connection towards the HS.	Step 1: Create a test scenario to simulate an internal event triggering the AG to initiate connection establishment. Step 2: Verify that the AG successfully establishes an ACL connection. Step 3: Create a test scenario to simulate a user-generated event triggering the AG to initiate connection establishment. Step 4: Verify that the AG successfully establishes an ACL connection. Step 5: Verify that the AG can establish an SCO link at any point after the ACL connection establishment. Step 6: Test both options described in Sections 4.2.1 and 4.2.2 to ensure they function correctly.
• 4.2.1	An in-band ring tone is an audible alert, such as a tone, melody, short music clip, that is transmitted by the AG, to the HS, to alert the user of an event; typically an incoming call. As shown in Figure 4.1, the AG may generate an in-band ring tone using the SCO	Yes ✓	HSP/HS/IAC/BV-02-I - HS: To verify that, if supported and provided by the AG, the HS gives an in-band ringing tone during connection establishment initiated from the AG. AG: To verify that, if supported and	Yes

Figure 3.6: The output or result page of the tool. The figure visualizes the results from analyzing the Bluetooth data set.

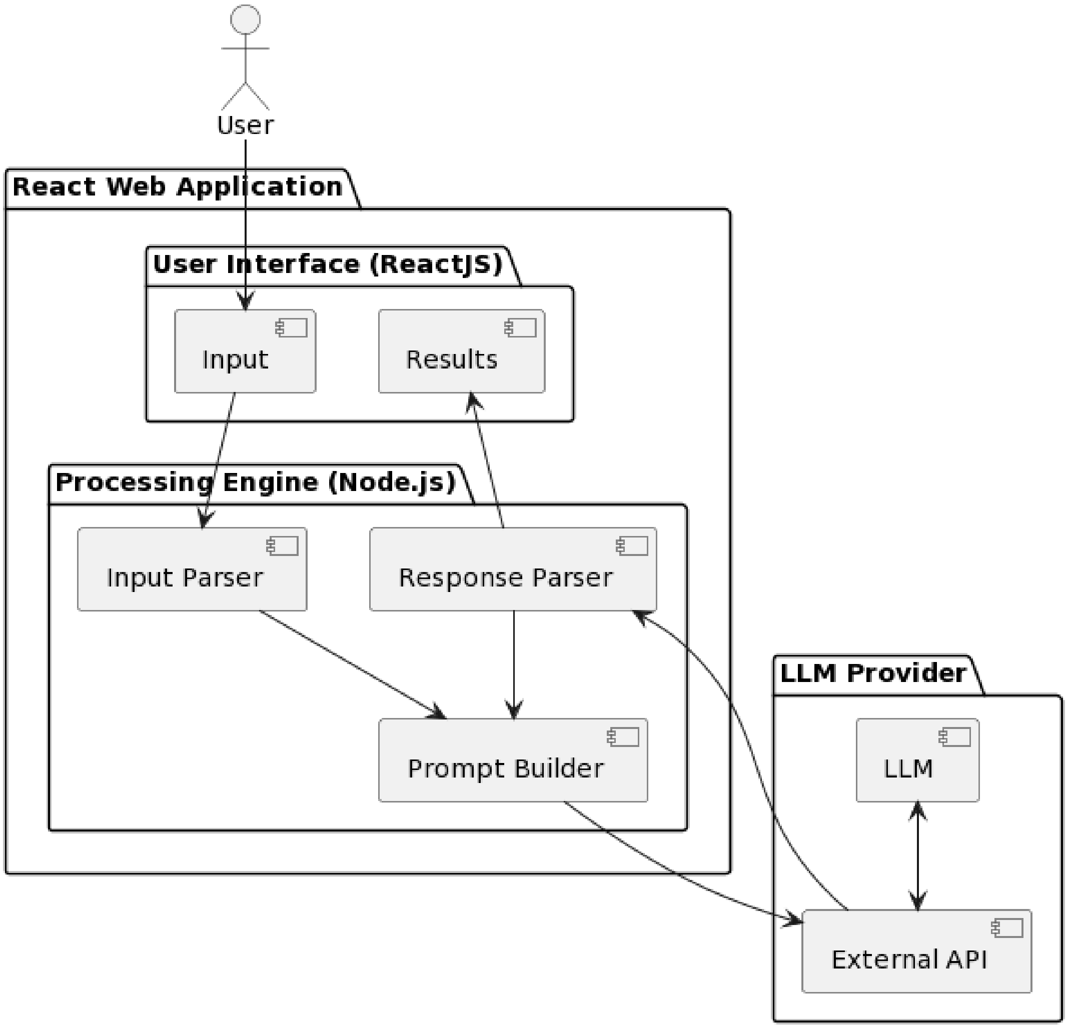


Figure 3.7: Architecture of the tool, with the various components and how they communicate.

4

Results and Analysis

Each section in this chapter corresponds to the research questions described in Section 1.1.1. These sections showcase the outcomes of each analysis and provide definitive answers to each research question.

4.1 RQ1: Relevant Requirements for a Testing Tool

The thematic analysis of the interviews has identified important features for a tool of this nature, which are reported in this section. Table 4.1 provides an overview of the themes that resulted from the thematic analysis. Each theme is also described in further detail, combined with support from interview quotes. We verified with our supervisor that the themes conform to the principle of theme orthogonality in the thematic analysis. This implies that the themes are connected, but they do not overlap with each other.

Challenges of Software Testing: Certain areas are typically problematic within software testing, such as negative or exceptional cases. This implies that it can be difficult to identify all the diverse and unique combinations for a test case, and not just limit testing to positive or normal cases.

> “All skilled developers can code the positive case, and they solve most of it. But that’s not where the danger lies, it’s in exception handling.” - P1

Verifying and validating a test suite is another challenging aspect of software testing. Once a set of tests is developed, it becomes difficult to determine if the testing is comprehensive enough. One of the participants shared an anecdote where a testing team assumed that they had executed all the essential tests, but in reality, the tests were not validated properly, rendering them useless.

> “Here, there is always uncertainty. When am I done? Have I covered everything that is needed?” - P2

The risks that come with software testing are crucial to consider, as inadequately testing a product or system for its intended purpose may result in severe consequences. Nevertheless, these risks also emphasize the significance and need for soft-

Table 4.1: Identified themes from the thematic analysis.

Theme	**Definition**
Challenges of software testing	Describes some common challenges professional testers face within the software testing process. For example, the negative cases, the risks of testing, and the verification or validation of test cases.
Quality aspects	This highlights the significance of various quality aspects required to ensure the completeness of a test suite, as well as how to validate this. For instance the importance of requirements, context, and traceability.
Testing levels	Emphasizes what level of testing the tool should be built for and where the tool is needed. Specifically, whether it is within unit-, integration-, or system-level.
Tool design	Describes different aspects that will need consideration when building the tool. Such as the input data, formatting, and when and how to receive the feedback.

ware testing in general.

> “In some cases, the company may no longer exist if this [testing] doesn’t go well.” - P1

Quality Aspects: In the previous theme above, *Challenges of Software Testing*, we highlighted the difficulties involved in verifying and validating a test suite. In this section, we will delve deeper into how the interviewees strive to ensure the completeness of the test suite and the essential characteristics of a good test suite, which have a strong connection to verification and validation. All three interviewees encountered difficulty in precisely defining these aspects.

> “[So there is no definitive answer to determine whether a test suite is good or not?] Not what I’ve found, no.” - P1

> “It bothers me a little that I answered so poorly about what a good test case is.” - P2

> “[What is a good test suite?] It’s very difficult.” - P3

Although providing a definite answer to what defines a good test suite was challenging, all interviewees brainstormed ideas that highlighted the crucial role of context. During their discussion of context, the interviewees pointed out that the significance of test cases varies depending on the product being tested. For instance, a car requires significantly more testing than a game developed for smartphones. This contrast highlights the challenge of determining whether a test suite is adequate or not.

> "This [a good test suite] depends entirely on context and the industry." - P1

One of the interviewees also mentioned the importance of asking questions to yourself, such as "What is the goal of me being a tester in this project?". All three interviewees concluded that the scope and requirements of a project also played an essential role in determining the characteristics of a good test suite and ensuring its completeness.

> "What is your scope? Then you have to check if you have fulfilled that scope." - P1

> "I usually dive into the requirements first and ask, have we tested this?" - P2

> "I try to find out what test basis is being used." - P3

In addition to the significance of scope and requirements, the interviewees also stressed the importance of requirements traceability or test alignment. This implies the ability to verify whether a requirement has been fulfilled or not, which is typically validated through test cases. If this can be quickly and consistently verified, it simplifies the testing process for the tester.

> "You try to enter the requirements and then write the tests and link the tests to the requirements so that you can measure progress." - P1

> "Which requirements are not covered at all? This relies on having requirement references all the way." - P2

> "It is important to ensure that the test cases are linked to requirements and such so that it is fully structured what it should be linked to and perhaps also explain the purpose of why it should be carried out in a certain way." - P3

Testing Levels: For this theme, we intended to focus on the level at which the tool would be most beneficial for the daily work of software testers. However, the interviewees had little to contribute except for the opinion that the tool would not be helpful at the unit level. Given their views on the significance of requirements, scope, and traceability, it makes more sense to target the system level, where this emphasis is more critical.

> "Unit is super uninteresting" - P1

> "I think the simplest type of tool is made at the unit test level. I think it's most needed at the system test level." - P2

Tool Design: The topic delves deeper into the essential considerations we must consider during the tool's construction. Additionally, it outlines the crucial requirements that the interviewees highlighted and potential issues that may arise. We initiate this theme by exploring whether the interviewees usually work to build new test suites or improve existing ones. However, both cases are common, and as we want to provide feedback to the testers regarding their test suite, we will primarily focus on improving existing tests.

> "I have worked in both scenarios. [both building and improving suites]" - P2

In one of the interviews, a participant recounted meeting someone who drew a parallel between the level of tools and assistance available to surgeons during an operation and the support that software testers should have while testing software. During another interview, a participant had a similar idea that with the help of an AI, you could get the support that testers currently lack.

> "He had thought of using something like Alexa, that he can talk to the computer and get answers back, like a surgeon" - P1

> "...so he wanted some computerized support, like a surgeon." - P1

> "But maybe there is some AI or something experience-based that says here we often fail, so you have some kind of database-like thing" - P2

Testing poses a challenge because the input data for the tool can vary significantly. Test cases will vary depending on the level of testing being conducted. For instance, unit testing primarily involves code, whereas system or acceptance testing may rely on natural language. The formats in which the tests are delivered may differ a lot as well. Taking the natural language case, the program in which the test is written may be in Microsoft Excel, Microsoft Word, or within a plain text file. This presents a challenge when we develop the tool.

> "I think you will need to be prepared to handle a variety of formats for the test cases, so that's where I think you will have a big challenge" - P3

Nevertheless, the challenge may be addressed by utilizing an AI, particularly a proficient LLM. Since LLMs can manage a vast range of inputs through understanding and interpreting the meaning of various texts. This has been seen in the works of [13]–[17] to exemplify. Nonetheless, the input data will still require formatting.

> "I think you need to focus on getting the right data and then being able to modularize the data into an engine that can be applied by some AI, and then the answer needs to be controlled in some way." - P1

> "Something [a tool] that makes it possible to extract the test cases and put them in a relatively neutral container and then review it that way to ensure that the feedback is correct." - P3

Considering this, we have discovered that the conditions are appropriate for leveraging a natural language model, and with suitable input data, we shifted our focus toward how to utilize the tool effectively. Based on the theme of *Quality Aspects*, we emphasized the scope, requirements, and traceability. By combining these aspects with our ideas on how the tool could execute a task, we formulated a concept for its construction. The participants also contributed ideas on how the tool could be utilized.

> “Now you have entered a lot of errors here, have you thought about these cases, I can’t see that you have thought about it, you may need to tighten the leaking boat, now you have only used this type of methodology when writing tests, there is an opportunity to add this methodology” - P1

> “as I mentioned earlier [output examples]; have you thought about performance testing? Have you thought about security testing?” - P1

> “I can imagine that you want help with: are there gaps in between the test cases, so that you look at the pile of test cases and match them against expectations in the form of a requirement mass in some way” - P2

Another dimension of this theme concerns the components that place greater emphasis on functional requirements. Initially, we sought to determine the appropriate environment for building the tool. However, the interviewees did not express a strong opinion on this matter.

> “I think that if I get help, I don’t care if it runs as a script or if it’s packaged for VSCode or Jira or something like that” - P1

> “Maybe you go for the really big market-leading things, or you don’t care about the market, and as you suggested; we make a standalone and then let the others adapt to it or adapt their process to it” - P2

An additional functional requirement we sought to address was when the tester desired to receive feedback during the testing process — specifically, before, during, or after making modifications to the test suite. Initially, the responses were quite varied, and the interviewees had some initial opinions. However, upon further reflection, they all began to lean towards receiving feedback after implementing their changes to verify the correctness of their modifications.

> “The best would have been before, it would be luxurious in real-time, but the most reasonable or easiest would be to get it afterward.” - P1

> “Now I’m probably in some mixture of during and after, maybe.” - P2

> “I prefer to get it before I submit it. So after I mark something as done, I like to have it analyzed.” - P3

RQ1: What are the relevant requirements of a tool that can aid testers in evaluating the quality of test artifacts?

The main requirements raised by practitioners in our interviews are the following:
- The tool should be built for system-level tests.
- The tool should be capable of processing various inputs, including manual tests in natural language and automatic tests.
- The inputs should be analyzed using a large language model.
- The resulting feedback should contain details on whether a test suite tests its corresponding requirements.
- The tool should be able to provide testing suggestions for any requirement.
- The tool should be capable of tracing the alignment between a requirement and its corresponding test case (if a corresponding test case exists).

The value that such a tool adds is assistance when evaluating whether a test case or suite fulfills a targeted set of requirements. It also offers a fresh perspective on a particular test case or suite and suggests potential improvements. Furthermore, it can assist in onboarding new projects, allowing testers to gain a comprehensive understanding of the value provided by the existing test suite. With these results from the thematic analysis, we are well-positioned to achieve desirable outcomes from the development and utilization of this tool. Below is exemplified use case scenario of the tester “Kim”.

Kim, an experienced software tester, has recently joined a new company and project. Kim’s assignment is to assess the existing test suite in relation to the requirement specification. However, Kim lacks insights into the software, making it difficult for Kim to determine the value provided by each test case. Despite Kim’s familiarity with the programming language used in both the software and the test suite, Kim struggles to establish the relevance of specific test cases to individual requirements. To aid Kim’s evaluation, Kim turns to our tool, which helps to confirm the value of each test case within the suite. With the tool’s assistance, Kim successfully identifies that some requirements are inadequately tested. Kim then proceeds to create new tests within the suite, which are analyzed using the tool. The tool verifies that the new tests adequately cover the remaining requirements.

4.2 RQ2: Opportunities and Challenges of Prompt Engineering

As the requirement from RQ1 stated, the tool should be capable of processing various inputs, including manual tests in natural language and automatic tests. The requirements for the tool also mention that the inputs should be analyzed using an LLM. See Section 4.1. To achieve this, we developed a proof of concept on prompt engineering, since this was how we utilized the LLM.

As mentioned in Section 3.1.2, this was done through a trial-and-error procedure, where we followed different sets of prompting techniques to generate the desired response. The response needed to be in a structured format, consistent and accurate when received. Meaning that the mapping of inputs, which included requirements and test cases, should consistently produce the same correct output for the same input. Table 4.2 represents the different prompts tested together with the corresponding prompting technique used. The prompts with IDs such as PRy.x refers to similar prompts that have been iterated.

Table 4.3 visualizes the inputs and outputs for the prompts. Note that the IDs in both tables, 4.2, and 4.3, represent the same set of prompts. The inputs in Table 4.3 are represented as a concatenation of prefixes, requirements, infixes, test cases, and suffixes. This implies that the prefix, infix, and suffix correspond to the manually created prompts, while the requirements and test cases originate from the two input files. The output describes how the generated response was perceived. "Consistent" refers to the generation of identical outputs across multiple instances. "Accurate" pertains to the ability of the language model to align the requirements with the corresponding test cases correctly, i.e. that the LLM correctly performs the task given. "Structured" denotes whether the LLM generated a response with a structured format that could be easily parsed by the tool when received. The prompts that contain chains of several prompts are also numbered in the table within the input column.

Table 4.2: The different prompts tested, with their corresponding ID as well as what prompting techniques were used.

ID	Prompting Techniques
PR1	Chaining LLM steps, Zero-shot prompting
PR2	Instruction with prompt, One-shot prompting
PR3.0	Role-Playing, Instruction with prompt, One-shot prompting
PR3.1	Role-Playing, Chaining LLM steps, One-shot prompting
PR3.2	Role-Playing, Chaining LLM steps, Instruction with prompt, One-shot prompting
PR4.0	Role-Playing, Chaining LLM steps, Instruction with prompt, Few-shot prompting
PR4.1	Role-Playing, Chaining LLM steps, Instruction with prompt, Few-shot prompting
PR5.0	Instruction with prompt, Few-shot prompting
PR5.1	Chaining LLM steps, Instruction with prompt, Few-shot prompting
PR6	Chain-of-Thought prompting, Few-shot prompting
PR7	Chain-of-Thought prompting, Chaining LLM steps, Few-shot prompting

Due to an agreement with TestScouts, we do not provide a fully exemplified view of each prompt, but to clarify the tables further we have provided the prompts PR1 and PR2 in the tables 4.4, 4.5, 4.6 and 4.7. The tables contain the input and three generations of outputs. We also motivate the classification of the prompts, as well as the quality of the output to provide a better understanding.

PR1 is classified as utilizing Chaining LLM steps and Zero-shot prompting. Chaining LLM steps involves the sending of multiple prompts that depend on the information provided within each prompt. In Table 4.4, we observe that the initial prompt sets the context for the following prompts, while the fourth prompt builds upon the

Table 4.3: The different prompts tested, with their corresponding ID as well as their input and output. Each prompt might include different parts from the input file, such as Requirements (Reqs), Test Cases (TCs), Prefix (Pfx), Infix (Ifx), and Suffix (Sfx).

ID	Input	Quality of the output
PR1	1st prompt: Pfx 2nd prompt: Pfx + Reqs 3rd prompt: Pfx + TCs 4th prompt: Pfx	Inconsistent, Relatively Accurate, Unstructured
PR2	Pfx + TCs + Ifx + Reqs + Sfx	Inconsistent, Inaccurate, Structured
PR3.0	Pfx + Reqs + Ifx + TCs + Sfx	Consistent, Inaccurate, Unstructured
PR3.1	1st prompt: Pfx 2nd prompt: Pfx + Reqs + Ifx + TCs + Sfx	Consistent, Inaccurate, Unstructured
PR3.2	1st prompt: Pfx 2nd prompt: Pfx + Reqs + Ifx + TCs + Sfx	Consistent, Inaccurate, Unstructured
PR4.0	1st prompt: Pfx 2nd prompt: Pfx + Reqs + Ifx + TCs + Sfx	Consistent, Inaccurate, Structured
PR4.1	1st prompt: Pfx 2nd prompt: Pfx + TCs 3rd prompt: Pfx + Reqs + Sfx	Consistent, Inaccurate, Structured
PR5.0	1st prompt: Pfx + TCs + Sfx 2nd prompt: Pfx + Reqs + Sfx	Consistent, Inaccurate, Structured
PR5.1	1st prompt: Pfx + TCs + Sfx 2nd prompt: Pfx + Reqs + Sfx 3rd prompt: Pfx 4th prompt: Pfx 5th prompt: Pfx	Inconsistent, Accurate, Structured
PR6	1st prompt: Pfx 2nd prompt: Pfx + TCs + Sfx 3rd prompt: Pfx + Reqs + Sfx 4th prompt: Pfx	Consistent, Relatively Accurate, Structured
PR7	1st prompt: Pfx + Reqs + Sfx 2nd prompt: Pfx + TCs + Sfx 3rd prompt: Pfx + Reqs + Ifx + TCs + Sfx	Consistent, Accurate, Structured

Table 4.4: This table visualizes the four prompts used for PR1, as well as three generated responses. The bold text within the table only highlights the different parts of the prompts as represented in Table 4.3, and is not part of the actual prompt.

PR1	
1st prompt	**Pfx:** I will give you three different prompts (starting after this one). The first one contains a list of several requirements for a software. The second is a test file that contains different test cases. The third will provide further instructions.
2nd prompt	**Pfx:** Here's the list of requirements: **Reqs:** (.CSV CONTENTS GETS INSERTED HERE BY THE TOOL)
3rd prompt	**Pfx:** Here's the test file containing test cases: **TCs:** (.CSV CONTENTS GETS INSERTED HERE BY THE TOOL)
4th prompt	**Pfx:** Now I want you to list the requirements that are covered within the requirements provided in the second prompt, based on the test cases provided in the third prompt. If requirements in the list are not covered, list them and give some examples of how you could they could be tested.
Output 1	Based on the provided test cases, I will identify the requirements that are covered and list any requirements that are not explicitly covered. Here are the requirements covered by the test cases: Covered Requirements: Requirement ID: 4.2 - Audio Gateway Initiated ACL Connection Establishment Requirement ID: 4.2.1 - Using In-Band Ringing Requirement ID: 4.4 - Headset Control Following Connection Establishment Requirement ID: 4.5 - Audio Connection Release Requirements Not Explicitly Covered: Requirement ID: 4.3 - Headset Initiated ACL Connection Establishment Example test case: Test the ability of the HS to initiate an ACL connection and establish a call/audio connection with the AG. Requirement ID: 4.9.1.1 - Connection establishment Example test case: Test the ability of both the AG and HS to initiate the connection establishment process and successfully establish a connection. Requirement ID: 4.9.1.2 - Connection release Example test case: Test the ability of both the AG and HS to release the connection and verify that the profile's serial port connection is also released. Requirement ID: 4.9.1.3 - Sniff mode Example test case: Test the behavior of devices in sniff mode during an ACL connection when a call is in progress and when a call is not in progress. These are just examples of how the requirements could be tested. The actual test cases would need to be more detailed and specific, covering various scenarios and conditions to thoroughly validate each requirement. Please note that the identification of covered and uncovered requirements is based on the information provided, and there may be additional requirements or specific details that need to be considered for a comprehensive analysis.

Table 4.5: Table 4.4 (continued).

PR1	
Output 2	Based on the test cases provided in the second prompt, here is a list of requirements that are covered: Requirement 4.2: Audio Gateway Initiated ACL Connection Establishment Covered by test cases: HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I, HSP/HS/IAC/BV-02-I, HSP/AG/IAC/BV-02-I Requirement 4.2.1: Using In-Band Ringing Covered by test cases: HSP/HS/IAC/BV-02-I, HSP/AG/IAC/BV-02-I Requirement 4.3: Headset Initiated ACL Connection Establishment Covered by test cases: HSP/HS/OAC/BV-01-I, HSP/AG/OAC/BV-01-I Requirement 4.4: Headset Control Following Connection Establishment Covered by test cases: HSP/HS/ACR/BV-01-I, HSP/AG/ACR/BV-01-I Requirement 4.5: Audio Connection Release Covered by test cases: HSP/HS/ACR/BV-01-I, HSP/AG/ACR/BV-01-I Requirement 4.9.1.1: Connection establishment Covered by test cases: HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I Requirement 4.9.1.2: Connection release Covered by test cases: HSP/HS/ACR/BV-01-I, HSP/AG/ACR/BV-01-I The following requirements are not covered by the provided test cases: Requirement 4.9.1.3: Sniff mode This requirement is not covered in the given test cases. To test this requirement, a test case could be created where the devices enter sniff mode while maintaining an active connection, and the functionality and power consumption in this mode are evaluated. Other unspecified requirements The test cases provided in the second prompt do not cover all the requirements mentioned in the first prompt. There may be additional requirements that are not addressed. To ensure complete coverage, additional test cases can be designed based on the remaining requirements, specifying the initial conditions, test procedures, and expected outcomes. It is important to note that the coverage of requirements depends on the context and specific needs of the software. The provided analysis is based solely on the information provided in the prompts.

Table 4.6: Table 4.4 (continued).

PR1	
Output 3	Based on the provided test cases, I will list the requirements covered from the first prompt. If any requirements are not covered, I will provide examples of how they could be tested: Covered Requirements: Requirement ID: 4.2 (Audio Gateway Initiated ACL Connection Establishment) Covered by test cases: HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I Requirement ID: 4.2.1 (Using In-Band Ringing) Covered by test cases: HSP/HS/IAC/BV-02-I, HSP/AG/IAC/BV-02-I Requirement ID: 4.3 (Headset Initiated ACL Connection Establishment) Covered by test cases: HSP/HS/OAC/BV-01-I, HSP/AG/OAC/BV-01-I Requirement ID: 4.4 (Headset Control Following Connection Establishment) Covered by test cases: HSP/HS/ACR/BV-01-I, HSP/AG/ACR/BV-01-I, HSP/HS/ACR/BV-02-I, HSP/HS/ACR/BV-01-I Requirement ID: 4.5 (Audio Connection Release) Covered by test cases: HSP/HS/ACR/BV-01-I, HSP/AG/ACR/BV-01-I Requirement ID: 4.9.1.1 (Connection establishment) Covered by test cases: HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I Requirement ID: 4.9.1.2 (Connection release) Covered by test cases: HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I, HSP/HS/ACR/BV-01-I, HSP/AG/ACR/BV-01-I Requirement ID: 4.9.1.3 (Sniff mode) Not covered by the provided test cases. Requirements not covered in the test cases: Requirement ID: 4.9.1.3 (Sniff mode) This requirement can be tested by introducing a scenario where the devices need to reduce power consumption while maintaining an active connection. The test can involve simulating an active call or non-call state and checking if the devices enter and exit sniff mode accordingly. Example test case for Requirement ID 4.9.1.3: Test Case ID: HSP/Sniff/BV-01-I Test Purpose: To verify that the devices can enter and exit sniff mode while maintaining an active connection. Reference: 4.9.1.3 (Sniff mode) Initial Condition: Both devices are initialized (see Section 3.2.2). The devices have an active audio connection established. Test Procedure: Simulate an active call/non-call scenario. Monitor the power consumption and behavior of the devices. Expected Outcome: The devices enter sniff mode to reduce power consumption while maintaining the active connection. The devices exit sniff mode when necessary, such as during AT command exchange or resuming the call. By including the additional test case and executing it, we can ensure coverage for Requirement ID 4.9.1.3 (Sniff mode).

information from the previous two prompts, exemplifying the concept of chaining LLM steps. The prompt is also classified as utilizing zero-shot prompting. This is because the LLM should perform a single task solely based on the information presented in the initial three prompts, without any explicit examples or instructions on the expected correct response. This characteristic is evident in the fourth prompt.

The output quality is characterized as inconsistent, inaccurate, and unstructured. Its inconsistency arises from the fact that the generated output varies with each occurrence. For example, the length and content of the output contain significant differences across the three generations. It also demonstrates a slight inaccuracy in its performance of the assigned task, leading to its relatively accurate classification. For instance, in Output 1, the LLM failed to specify any test cases altogether. In Output 3, the inclusion of Requirement ID: 4.9.1.3 is contradictory, being listed as both covered and not covered. Nonetheless, aside from these instances, the mapping is correct. Moreover, the outputs exhibit an unstructured nature. This is evident as the responses consist of lengthy texts with varying content from one prompt to another. Such variability makes it challenging to extract essential information from the generated output, contributing to its classification as unstructured.

Table 4.7: This table visualizes the prompt used for PR2, as well as three generated responses. The bold text within the table only highlights the different parts of the prompts as represented in Table 4.3, and is not part of the actual prompt.

PR2	
1st prompt	**Pfx:** Here is a test file in the .csv format: .csv test cases: **TCs:** (.CSV CONTENTS GETS INSERTED HERE BY THE TOOL) **Ifx:** For this test file, I have a few requirements which are also in the .csv format: .csv format requirements **Reqs:** (.CSV CONTENTS GETS INSERTED HERE BY THE TOOL) **Sfx:** Now I only want you to answer with a simple yes or no. However, if you answer no then I want you to print what requirement is not fulfilled. Here is my question to you (I only want you to answer this question): Are the requirements fulfilled with the provided test file?
Output 1	No, the requirements are not fulfilled with the provided test file.
Output 2	No, the requirements are not fulfilled with the provided test file.
Output 3	Yes, the requirements are fulfilled with the provided test file.

PR2 is classified with the techniques Instruction with prompt and One-shot prompting. Both these techniques are quite similar per definition which is illustrated here. Instruction with prompt refers to the instructions and rules set for the generated output and one-shot prompting refers to giving one example of what is deemed correct. This is evident in the Sfx part of the prompt, as seen in Table 4.7. More specifically, the sentence “Now I only want you to answer with a simple yes or no.” since it provides guidance for the generated response.

The quality of the output is deemed as Inconsistent, Inaccurate, and Structured. While only generating three different prompts, Output 3 is not the same as the first two which makes it inconsistent. Moreover, even though Output 1 and Output 2 are correct, the LLM did not provide the unfulfilled requirement(s) and answered with more than ´´Yes” or “No”, which was specified in the prompt. Therefore, we deem it inaccurate. However, we do observe an improvement in the structure, as the important information can easily be extracted from the generated response.

To test these techniques, the Bluetooth sample data was used. Furthermore, we will discuss the opportunities and challenges associated with each technique. Note that more than one technique was always used. This is because zero-, one-, and few-shot prompting indicates the number of exemplified outputs provided to the LLM. Several other techniques were often combined as well, due to overlapping definitions. To clarify how the different techniques were combined, see Table 4.2.

Zero-shot prompting: The main challenge encountered with Zero-shot prompting was achieving consistent and structured output. Since there were no predefined examples of how the LLM should respond, the generated outputs had significant variations. It produced lists on some occasions, plain text on others, and occasionally provided simple yes or no answers. Using Zero-shot prompting allows the LLM to have unrestricted thinking as the prompt does not provide clear instructions on how to respond. This means that even vague prompts can be answered, but the accuracy and reliability of those answers may vary. However, in the case of the tool's use, this technique was not suitable.

One-shot prompting: By adding just a single example of how the LLM should respond, we observed an improvement in both consistency and structure of the generated response. This proved useful as the desired response for the tool was required to be in a specific format. Even though the output was in a proper format, it often showed tendencies of being inaccurate. In other words, if the prompt was not accurately specified, even if the answer provided aligns with the examples or instructions given, it may still be incorrect in reality. We also observed that given a specified example, the output generated was often disconnected from the context of the prompt. For example, using a set of requirements and test cases as part of the prompt sometimes generated responses with newly fabricated requirements and test cases.

Few-shot prompting: Adding even more examples and more concretely defining the output format, we saw an even greater improvement in the consistency and structure of the generated output, compared to One-shot prompting. Adding more context, examples, and rules to the prompt made it longer and more complex. However, we noticed that the increased complexity caused the LLM to forget important instructions and context. For instance, when there were multiple instructions and examples for generating output, it frequently missed including all the requirements and test cases, resulting in reduced accuracy. This was also observed to be a common theme for almost all of our prompts.

Instruction with prompt: This technique shares significant similarities with Few-shot prompting, leading to similar outcomes. Specifically, when the format of the desired output is explicitly specified, it tends to improve the structure and consistency of the generated responses. However, by adding more complexity to the prompts, the accuracy got impacted negatively. We also noticed that when creating prompts with several instructions and rules, it's easy to create contradicting prompts. To exemplify, with the prompt "I only want you to answer with a simple yes or no. However, if you answer no then I want you to... [insert task]". With this prompt, it is difficult to know which instruction the LLM will follow. Because if the answer is no and the LLM should perform a second task, the response will no longer be a simple yes or no, which causes a contradiction.

Role-Playing: Role-playing initially showed potential. We wanted the LLM to immerse itself as a software tester and define its output in regard to this. The LLM also improved accuracy as it showed signs of being more detailed. Although the generated output was fairly accurate, it was also rather unstructured and inconsistent. When trying to resolve this, the prompts got more complex and thus reducing the accuracy again.

Chaining LLM steps together: When employing chaining, the prompts resembled natural conversations, making them easier to define and write, reducing each prompt's complexity. However, despite the expectation that the language model would remember previous prompts within a conversation, we observed difficulties in its ability to recall the content of those prompts. This limitation could be attributed to the complexity of the prompts themselves.

Chain-of-Thought prompting: This technique showed the best results for the challenge when the prompts become long and complex. By dividing the main task, to align the requirements and test cases, into several steps, you could still provide the LLM with several instructions on how the format should look. As each step becomes more focused and manageable, we observed significant improvements in the generated output's structure, consistency, and accuracy. Additionally, the output generated by the language model can be utilized within our tool to create new prompts, further enhancing the capabilities and adaptability of the system.

To summarize, we will also present the exemplified view of PR7 (see Table 4.8 and 4.9), which represents the final prompt utilized by the tool in its current state. The prompting techniques employed include: *Chain of thought prompting*, *Chaining LLM steps together*, and *Few-shot prompting*. It is also important to note that the generated output comprises only one requirement and its corresponding test case. This is due to the fact that the 3rd prompt exclusively contains a single requirement while searching for its corresponding match. Consequently, the 3rd prompt is executed multiple times based on the number of requirements present in the .csv file. However, as shown in the three generated outputs, the result contains all three desired quality attributes. Meaning that the output is consistent, accurate, and structured.

Table 4.8: This table visualizes the prompt used for PR7, as well as three generated responses. The bold text within the table only highlights the different parts of the prompts as represented in Table 4.3, and is not part of the actual prompt.

PR7	
1st prompt	**Pfx:** I am about to give you the string representation of the contents of a file. The file contains a requirement. The headers of the file may suggest that the data in the file is not a requirement, but I want you to think of it as requirement anyways, and that is VERY important. The file is in csv format. The file is called “requirements.csv”. What I want you to do is to find the ID of the requirement. The ID can be a name or a key of some sort. I also want you to find the description of the requirement. I also want you to find any other useful information about the requirement. Now I am going to give you the contents of the file and the only thing I want you to answer is in the following form: {“requirementID”: “[insert id here]”, “desc”: “[insert description here]”, “other”: “[insert other information here]”} The contents of “requirements.csv” are: **Reqs:** (CSV CONTENTS GETS INSERTED HERE BY THE TOOL) **Sfx:** I am going to parse your input in my javascript program, therefore, ONLY ANSWER IN THE FORM I GAVE YOU AND CHANGE " TO ’ IF IT IS INSIDE QUTATION MARKS.
2nd prompt	**Pfx:** I am about to give you the string representation of the contents of a file. The file contains a requirement. The headers of the file may suggest that the data in the file is not test cases, but I want you to think of them as test cases anyways, and that is VERY important. The file is in csv format. The file is called “test.csv”. What I want you to do is to find the ID of the test case. The ID can be a name or a key of some sort. I also want you to find the description of the test case. I also want you to find the steps of the test case, if there are any. If there is no information about steps in the test case, just think “No test steps provided”. I also want you to find the expected outcome of the test case, if there is any. Now I am going to give you the contents of the file and the only thing I want you to answer is in the following form: {“ID”: “[insert id here]”, “desc”: “[insert description here]”, “steps”: “[insert steps here]”, “expected_outcome”: “[insert expected outcome here]”} The contents of “test_case.csv” are: **TCs:** (CSV CONTENTS GETS INSERTED HERE BY THE TOOL) **Sfx:** I am going to parse your input in my javascript program, therefore, ONLY ANSWER IN THE FORM I GAVE YOU AND CHANGE “ TO ’ IF IT IS INSIDE QUTATION MARKS.

Table 4.9: Table 4.8 (continued).

PR7	
3rd prompt	**Pfx:** I have this requirement: **Reqs:** (THE TOOL INSERTS A REQUIREMENT HERE) **Ifx:** Would you say that any of the test cases in the file “tests.csv” are testing the requirement? If yes, answer ONLY with the test case ID(s) that are testing the requirement in the following form: {“requirementID”: “[insert requirement id]”, “tests”: “[insert test id 1], [insert test id 2], [insert test id 3], ...”} DO NOT ADD ANY TEXT BEFORE OR AFTER THE CURLY BRACKETS. If no, answer ONLY in the following form: {“requirementID”: “[insert requirement id]”, “tests”: “”} The contents of “tests.csv” are: **TCs:** (THE TOOL INSERTS TEST CASES HERE) **Sfx:** I am going to parse your input in my javascript program, therefore, ONLY ANSWER IN THE FORM I GAVE YOU.
Output 1	{“requirementID”: “4.2 Audio Gateway Initiated ACL Connection Establishment”, “tests”: “HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I, HSP/HS/IAC/BV-02-I, HSP/AG/IAC/BV-02-I”}
Output 2	{“requirementID”: “4.2 Audio Gateway Initiated ACL Connection Establishment”, “tests”: “HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I, HSP/HS/IAC/BV-02-I, HSP/AG/IAC/BV-02-I”}
Output 3	{“requirementID”: “4.2 Audio Gateway Initiated ACL Connection Establishment”, “tests”: “HSP/HS/IAC/BV-01-I, HSP/AG/IAC/BV-01-I, HSP/HS/IAC/BV-02-I, HSP/AG/IAC/BV-02-I”}

RQ2: What are the challenges and opportunities associated with prompt engineering for assessing and aligning tests with requirements?

When utilizing an LLM for complex and specific tasks, attaining both consistency and accuracy posed the greatest challenge. However, through the combination of the prompting techniques *Chain of thought prompting*, *Chaining LLM steps together*, and *Few-shot prompting*, we discovered that it is indeed possible to achieve this dual objective. By leveraging these techniques, new opportunities emerge for developing tools that can automate the labor-intensive task of aligning requirements to test cases. This, in turn, can enhance productivity and efficiency by relieving manual efforts in the software development process.

4.3 RQ3: Feasibility and Performance of the Testing Tool

The answer to this research question is derived from the following two subsections that examine the feasibility of a tool like this. Through the successful development of our tool and prompts and the promising results presented later in this section, we have unveiled that it is possible to develop an automated process for assessing and aligning tests with requirements.

4.3.1 RQ3.1: The Various Costs of the Tool

Financial Cost

The financial cost of using the tool depends entirely on the model used. Since we used OpenAI's API to access their GPT-3.5 model, each request costs money. Currently, the cost of using their API is \$0.002/1000 tokens (one token is roughly 4 characters of text).

Our prompts use hard-coded headers that look the same for each prompt, which means that the cost of the header for a prompt will always be the same. However, this is not the case for the data extracted from the CSV file since the requirement or test case description will vary in length. To circumvent this, we calculated the number of tokens for each Bluetooth requirement and test case mentioned in Section 3.1.2, as well as 8 requirements and 10 test cases from the Mozilla dataset and took the average of that. We only used 8 requirements and 10 test cases from the Mozilla dataset since we wanted an equal amount of requirements and test cases from both datasets. See Tables 4.10 and 4.11.

The table also shows the average number of tokens for the response objects (the responses from the language model) since the response tokens are included when OpenAI calculates the cost. These averages could then be used to calculate the average sum of each requirement- and test case prompt. The "requirement is tested" prompt is a bit more complicated since it has more than one factor affecting the prompt's length. Most of the prompt can be calculated by taking the sum of the header and the average of the requirement response object. The sums can be seen in Table 4.12.

The rest of the prompt varies depending on how many test cases you give the tool. If you, for example, give it 20 test cases, then the cost of that section will be:

$$20 * test\_case\_response\_object\_average = 20 * 148 = 2960\ tokens$$

Since OpenAI's API has a limit[1] on how many tokens you can send the model at a time, we had to limit how many test cases to include per *Requirement is tested* prompt. Our findings indicate that using 20 test cases per prompt was sufficient.

---

[1] `https://platform.openai.com/docs/models/gpt-3-5`

Table 4.10: Token counts of the requirements. The requirements with IDs that start with a “4” are from the Bluetooth dataset and the rest are from the Mozilla dataset.

	Token count		
ID	**CSV**	**Response object**	**Is tested response object**
4.2	82	91	45
4.2.1	125	122	40
4.3	259	254	33
4.4	108	113	59
4.5	123	125	56
4.9.1.1	72	80	92
4.9.1.2	89	94	59
4.9.1.3	76	83	22
Req_001	279	47	18
Req_002	281	49	22
Req_003	278	46	22
Req_004	278	46	18
Req_005	281	50	15
Req_006	282	50	22
Req_007	276	44	30
Req_008	284	51	15
Average	**198.31**	**84.06**	**35.50**

Table 4.11: Token counts of the test cases. The test cases with IDs that start with "HSP" are from the Bluetooth dataset and the rest are from the Mozilla dataset.

	Token count	
ID	**CSV**	**Response object**
HSP/HS/IAC/BV-01-I	324	253
HSP/AG/IAC/BV-01-I	324	253
HSP/HS/IAC/BV-02-I	273	209
HSP/AG/IAC/BV-02-I	273	187
HSP/HS/OAC/BV-01-I	197	92
HSP/AG/OAC/BV-01-I	164	60
HSP/HS/ACR/BV-01-I	168	105
HSP/AG/ACR/BV-01-I	172	109
HSP/HS/ACR/BV-01-I	185	123
HSP/HS/ACR/BV-02-I	157	89
view-bookmarks	576	236
add-bookmark	470	126
file-bookmark	518	161
create-bookmark	505	149
delete-bookmark	510	168
add-delete-ptoolbar-bmark	528	187
create-folder	612	257
surf-bookmark	507	164
surf-all-bookmarks	408	91
edit-properties	556	204
Average	**371.35**	**161.15**

Table 4.12: Token count of the requirement prompt, test prompt, and requirement is tested prompt without including the test cases.

Section of Prompt	Tokens	Price ($)
Requirement header	182.00	$3.64 * 10^{-4}$
Example requirement average	198.31	$3.97 * 10^{-4}$
Response object average	84.06	$1.68 * 10^{-4}$
Requirement total average	**464.38**	$\mathbf{9.29 * 10^{-4}}$
Test header	295.00	$5.90 * 10^{-4}$
Example test case average	371.35	$7.43 * 10^{-4}$
Response object average	161.15	$3.22 * 10^{-4}$
Test total average	**827.50**	$\mathbf{16.55 * 10^{-4}}$
Requirement is tested header	171.00	$3.42 * 10^{-4}$
Requirement average	84.06	$1.68 * 10^{-4}$
Response object average	35.50	$0.71 * 10^{-4}$
Requirement is tested total (without test cases)	**342.00**	$\mathbf{6.84 * 10^{-4}}$

In other words, in the scenario where 20 test cases are provided as input, only one *Requirement is tested* prompt needs to be sent per requirement. However, if the number of test cases exceeds 20, multiple *Requirement is tested* prompts per requirement are necessary. The resulting cost of the prompt with this example can be seen in Table 4.13.

Table 4.13: Token count of the requirement is tested prompt using 20 test cases.

Section of Prompt	Tokens	Price ($)
Requirement is tested total (without test cases)	342.00	$6.84 * 10^{-4}$
Test cases	3223.00	$64.46 * 10^{-4}$
Requirement is tested total	**3565.00**	$\mathbf{71.30 * 10^{-4}}$

Having an average token count for all prompts makes it possible to calculate the total token count for running an analysis with the tool. An example of an analysis with the tool using 20 requirements and 20 test cases can be calculated by the following formula:

NR: Number of requirements
NT: Number of test cases
RA: Requirement tot average
TA: Test tot average

RTT: Requirement is tested tot

$$NR*RA+NT*TA+NR*RTT = 20*464.38+20*827.50+20*3565.00 = 97137.60\,tokens$$

The cost in US dollars for running an analysis with 20 requirements and 20 test cases will then be:

$$total\_tokens * token\_cost = 97137.60 * \frac{0.002}{1000} = \$0.19$$

Since both the number of requirements and the number of test cases affect how many *Requirement is tested* prompts will be sent, the increase in tokens will not be linear when increasing the input size. For instance, doubling the input size (40 requirements and 40 test cases) results in a token count of approximately 336,875, with a corresponding cost of approximately \$0.67. This represents a 3.53 times increase in cost, as shown in equation 4.1.

$$\frac{0.62}{0.176} = 3.52 \tag{4.1}$$

When the input size is increased by a factor of 10, the token count reaches approximately 7,388,375, which incurs a cost of approximately \$14.78. This represents a 77.79 times increase in cost.

Time Cost

There are several time costs of the tool, such as the time spent on learning how to use the tool, preparing the requirements- and test suite files, maintenance time, and the actual time spent on using the tool. In this analysis, we will focus on evaluating the time cost associated with the tool's usage. Other aspects, such as learning time, preparation, and maintenance time, will be discussed in detail in the subsequent Discussion chapter.

Considering the tool's simplicity, the analysis time is the primary component of the usage time. Analysis time refers to the duration from when the user initiates the analysis by pressing the "Run Analysis" button until the results are displayed on the screen. It is important to note that the workload on OpenAI's servers significantly influences the tool's usage time. The responsiveness of each prompt depends on the current server load, impacting the overall speed of the tool's operation.

The overall usage time of the tool comprises the analysis time and the time required for the user to load the requirements and test suite files. Table 4.14 presents the analysis time and total time for three different runs, each consisting of 49 requirements and 44 test cases.

The table shows that the average usage time for this input is approximately 12 minutes, out of which roughly 6 seconds are spent by the user on inputting the files into

Table 4.14: Analysis times of three different runs, each with 49 requirements and 44 test cases.

Run	Analysis time	Usage time
1	10 minutes and 58.01 seconds	11 minutes and 2.81 seconds
2	12 minutes and 37.44 seconds	12 minutes and 40.48 seconds
3	11 minutes and 49.06 seconds	11 minutes and 59.21 seconds

the tool. The time spent by the user will naturally vary depending on how quickly they are able to locate their files. Although the runs utilized the same requirements and test cases, there was considerable variation in the resulting analysis time. This discrepancy can be attributed to the fact that the runs were not conducted simultaneously, resulting in differing workloads on OpenAI's servers.

RQ3.1: What are the costs of performing such an analysis?

The **financial cost** of utilizing this tool remains reasonable for smaller input sizes. However, as the input size increases, the cost escalates non-linearly, making it progressively more expensive. When analyzing a set of 200 requirements and 200 test cases, we found an approximate cost of $13.64 for the Bluetooth example.
The **time cost** follows a similar pattern as the time to receive an output (in minutes) increases with the number of prompts, resulting in additional time spent waiting for responses from the language model. It is worth mentioning that the usage time of the tool may vary between runs with the same input due to varying workloads on OpenAI's servers. However, on average, the tool requires approximately 12 minutes of usage time for an input size of 49 requirements and 44 test cases.

4.3.2 RQ3.2 - The Accuracy, Recall, and Precision of the Tool

We used three different metrics to assess the tool. The first metric, **Accuracy**, measures the tool's accuracy in determining whether a requirement is tested. It evaluates how effectively the tool identifies whether a requirement is covered by a test case. The accuracy could be calculated by counting the number of TPs, FPs, TNs, and FNs and using Formula 4.2. In this case, a TP refers to when a requirement is correctly marked as tested. A TN refers to when a requirement is correctly marked as not tested. An FP refers to when a requirement is incorrectly marked as tested, and lastly, an FN refers to when a requirement is incorrectly marked as not tested. The results from runs 1, 2, and 3 are shown in Table 4.15.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} * 100 \tag{4.2}$$

Table 4.15: The resulting accuracy when running the tool using 49 requirements and 44 test cases from the Mozilla dataset.

Run	**True positives**	**False positives**	**True negatives**	**False negatives**	**Accuracy**
1	42	5	0	2	85.71
2	41	4	1	3	85.71
3	42	4	1	2	87.76

The second metric, **Recall**, assesses the tool's average recall in mapping test cases for each requirement. It focuses solely on the correct mapping of test cases, regardless of any incorrect mappings to the same requirement. To determine this metric, we counted the occurrences of TPs, FPs, TNs, and FNs for each requirement. It's important to note the distinction: the accuracy metric calculates the total number of TPs, FPs, TNs, and FNs for the entire analysis, whereas the recall metric counts the number of TPs, FPs, TNs, and FNs for each requirement individually to compute the recall for that specific requirement. When assessing the average recall a TP refers to when a requirement is aligned with the correct test case, and an FN refers to when a test case is not included in the alignment, when it should. The recall for each requirement can be calculated using Formula 4.3.

$$Recall = \frac{TP}{TP + FN} * 100 \tag{4.3}$$

The third metric, **Precision**, evaluates the precision of the tool in mapping test cases to the correct requirement while also considering if it is mapping any incorrect test cases to a specific requirement. It provides a measure of how well the tool identifies the appropriate mappings while avoiding incorrect associations and it was evaluated in a similar manner to the recall. In addition, TP, TN, FP, and FN are the same as in the average recall assessment. To calculate the precision for each requirement we used Formula 4.4.

$$Precision = \frac{TP}{TP + FP} * 100 \tag{4.4}$$

The total recall and precision for each run is the average of all recalls and precisions. The results for runs 1, 2, and 3 are shown in Table 4.16.

To assess the performance of the tool, it is necessary to compare the three different metrics that are presented. As shown in Table 4.15 and 4.16, the accuracy and recall percentages are relatively high compared to the precision. Since accuracy is based

Table 4.16: The table shows the number of TPs, FPs, TNs, and FNs for some of the requirements for runs 1, 2, and 3. The table also shows the recall and precision for the requirements along with the average recall and precision for the runs. 49 requirements and 44 test cases from the Mozilla dataset were used for all runs.

Requirement ID	**True positives**	**False positives**	**True negatives**	**False negatives**	**Recall**	**Precision**
Run 1						
1	1	2	41	0	100.00	33.33
2	1	16	27	0	100.00	5.88
3	1	4	39	0	100.00	20.00
...						
42	0	1	42	1	0.00	0.00
43	0	2	41	1	0.00	0.00
...						
48	1	0	43	0	100.00	100.00
49	0	1	43	0	100.00	0.00
Average					**85.71**	**46.67**
Run 2						
1	1	2	41	0	100.00	33.33
2	1	16	27	0	100.00	5.88
3	1	2	41	0	100.00	33.33
...						
42	0	1	42	1	0.00	0.00
43	0	2	41	1	0.00	33.33
...						
48	1	0	43	0	100.00	100.00
49	0	1	43	0	100.00	0.00
Average					**83.67**	**46.90**
Run 3						
1	1	2	41	0	100.00	33.33
2	1	16	27	0	100.00	5.88
3	1	2	41	0	100.00	33.33
...						
42	0	1	42	1	0.00	0.00
43	1	1	42	0	0.00	50.00
...						
48	1	0	43	0	100.00	100.00
49	0	2	42	0	100.00	0.00
Average					**89.80**	**43.17**

on a binary result (i.e., “Yes” or “No”) and whether it is correct or not, it does not account for the correct alignment. Consequently, this measure could be considered unreliable, as it may potentially produce misleading results. However, from a use case perspective, the end user may shift focus from requirements that are deemed as tested and prioritize their time on requirements that are not adequately tested.

The recall percentage differs from accuracy as it considers alignment between test cases and requirements. However, it is important to note that when multiple test cases are mapped to the same requirement, this can potentially lead to ambiguity. It becomes challenging to determine which among these test cases is the “correct” one(s), making the recall measure potentially misleading. The term “correct” denotes the alignment that has been previously established for the dataset by a human.

Furthermore, the precision metric is considered the most reliable and comprehensive measure for evaluating the tool. It takes into account the most relevant factors and provides an accurate percentage without any room for misinterpretation with this dataset. This is due to the alignment ratio between requirements and test cases being 1:1. However if the dataset would contain multiple test cases for a single requirement, the precision percentage can become somewhat misleading. This is because precision solely focuses on the correctness of the included test cases, rather than considering whether all the correct test cases for a requirement have been included.

RQ3.2: How accurate are those analyses?

The effectiveness of this tool relies on your specific needs. It can be a highly valuable resource if you’re seeking to identify tested and untested requirements. We found an average accuracy rate of 86.394%, an average recall rate of 86.394% on average, and an average precision rate of 45.582% for the Mozilla dataset.

In addition to the results provided within this chapter, our colleagues at TestScout were granted the opportunity to explore and test our tool during a workshop session. With the freedom to experiment and provide informal feedback, they were able to identify the limitations of the tool. However, it was encouraging to observe that several participants recognized the inherent opportunities and vast potential that a tool of this nature holds. Their insightful perspectives shed light on the significance and applicability of our innovative solution in the testing domain.

5

Discussion

This chapter delves into various factors that could have influenced our results, identifies potential threats to the validity of our Master's Thesis, and explores the potentials and limitations associated with the developed tool. By thoroughly examining these aspects, we aim to provide a comprehensive understanding of the contextual factors, challenges, and opportunities that surround our research findings and the tool's functionality. This analysis contributes to a more robust evaluation of the overall study and enhances the transparency and reliability of our conclusions.

The subsequent sections will offer detailed explanations expanding on the results presented in Chapter 4. We will delve deeper into noteworthy aspects of the results and provide explanations regarding specific topics related to the findings. Each subsection is dedicated to addressing its corresponding research questions.

5.1 RQ1: Requirement Elicitation

The outcome of RQ1 relies on the identification of specific codes and themes derived from transcribed interviews, analyzed through a thematic analysis. As a result, our interpretations of the participants' answers during the interviews inherently influenced the requirements elicited. Acknowledging that our understanding and interpretation may have unintentionally overlooked significant aspects of software testing that may require greater assistance and support is crucial. Furthermore, if this Master's thesis were to be replicated with a fresh set of participants, it is plausible that new and distinct requirements could emerge, reflecting the diverse perspectives and experiences of the new participants.

While the thematic analysis employed in this study captures the essential elements of each interviewee's responses, it is important to acknowledge that it may not fully capture the depth and intricacies of the complete interviews. Due to the nature of a thematic analysis, the comprehensive and nuanced content of the interviews, which spanned nearly an hour, is inevitably condensed and presented in a summarized form within the results. As a result, some of the richness and contextual details embedded within the participants' extensive answers may not be fully represented.

5.2 RQ2: Prompt Engineering

The various prompting techniques investigated in this Master's Thesis are just a small subset of numerous techniques that have emerged within the topic of prompt engineering. As the models and research in this area are still in their early stages, there is no definitive answer regarding the most optimal prompt technique. The selection of techniques examined in this study was motivated by their common usage and perceived suitability based on available literature. It is important to acknowledge that other prompting techniques may be more suitable for the developed tool. Additionally, considering our specific use case for the LLM, some techniques that were tested and discarded may be more effective for other tasks and for different LLMs. Therefore, when reviewing the results for RQ2, it is important to note that they are specific to our scenario, which aims to achieve high consistency, accuracy, and structural alignment between requirements and test cases.

5.3 RQ3: Evaluation of the Tool

During our evaluation of the tool, we observed that the cost of the tool increases at a rate greater than linear, which could potentially pose challenges when analyzing a large number of requirements and test cases. Nevertheless, it is worth noting that a cost of $13.64 for automating the task of mapping 200 requirements and 200 test cases that would otherwise be performed manually may still be deemed reasonable for many companies. See this example in more detail in Section 4.3.

Our evaluation of the tool's costs only considers the financial and time usage costs. However, it is essential to keep in mind other substantial expenses. When considering the development of the tool as a business solution, it's important to consider the financial and time costs associated with its maintenance for stakeholders. Furthermore, end-users should take into account the time required to prepare files for the tool and the learning curve associated with using it, as these costs can affect the overall usefulness of the tool.

As LLMs are a rapidly advancing technology, our tool has the advantage of flexibility in terms of the models it utilizes. This means that as new and improved models are released (for example GPT-4[1]), there is the potential for the tool to enhance its overall accuracy. While some maintenance costs may be associated with this process, such as refining prompts, it is expected to require minimal effort.

Based on the current prompts and language model of the tool, it appears to be quite accommodating, which leads to it sometimes generating an excessive number of test cases for a given requirement, in other words, it is prone to false positives. Furthermore, the accuracy results were only tested on one dataset with nearly 100% requirement coverage, which may lead to overestimated accuracy. Additionally, the tool's accommodating nature could potentially result in lower accuracy on datasets with less coverage.

---

[1] `https://openai.com/product/gpt-4`

Although the tool may not always accurately map the correct test cases to a specific requirement, this does not necessarily mean that the prompts or language model are at fault. It is possible that the descriptions for the requirements and test cases are too vague. In such cases, the tool could be used to double-check the current requirement and test case mapping. This may reveal that the requirements or test cases are not specific enough, making it a potential use case for the tool.

5.4 Threats to Validity

The purpose of this section is to highlight the potential threats to validity, allowing the reader to assess the reliability and credibility of the presented results. Following the guidelines proposed by Runeson and Höst [38], the threats to validity are categorized into four subsections: Construct Validity, Internal Validity, External Validity, and Reliability. With this structure, we comprehensively examine the various aspects that may impact the validity of the study's findings.

5.4.1 Construct Validity

Construct validity refers to the extent to which the analysis conducted accurately reflects the intended objectives and goals of the study. This mainly revolves around whether the five research questions have been adequately addressed. However, verifying this can be challenging due to the utilization of the Design Science research methodology. This approach involves continuous refinement and iteration of the developed artifact throughout the three cycles. Given that the requirements for the tool were elicited in the first cycle, the overall purpose and objectives of the tool might have been different if alternative requirements had been identified. Furthermore, if it had been determined that prompt engineering was not a suitable solution, the tool's development may have leaned more toward an algorithmic and rule-based approach. Consequently, this would have impacted the results for RQ2, RQ3, RQ3.1, and RQ3.2.

Furthermore, the results for RQ2 are somewhat lacking in specificity, suggesting that this aspect of the thesis could have been approached in a more empirical manner. By doing so, we could have provided more comprehensive insights into the distinctions among various prompting techniques in terms of consistency, accuracy, and structure, while also presenting the findings in a more apparent manner. However, given the scope of the thesis, our primary focus was on developing the tool, which required allocating more time to this aspect rather than conducting an extensive evaluation of alternative methods that could have been utilized for tool development.

5.4.2 Internal Validity

Internal validity refers to potential risks that threaten the accuracy and reliability of data collection processes and, consequently, the validity of the derived results. One potential threat that may have happened during the interviews for RQ1, is that participants may have shared the details of the interviews with one another. This is particularly problematic when participants are employees, as they may disclose confidential information to other employees, compromising the integrity of the interviews. Additionally, it is important to acknowledge that the requirements elicitation process relied on a limited number of interviews, specifically three. This implies that there is a possibility that important aspects may have been overlooked, which may have been identified if a larger number of interviews were conducted.

In order to conduct the analysis for RQ3, we used a specific dataset, namely the requirements and test cases derived from the Mozilla documentation. This data was fairly aligned in terms of the requirements to test case ratio. Meaning that the majority of the requirements had a corresponding test case. If we were to use a dataset where the requirements didn't have aligned test cases, the tool's accuracy results may have been affected.

5.4.3 External Validity

External validity concern the extent to which the results of a study can be generalized or applied to broader settings, or contexts. The requirements identified from RQ1 have led to a relatively narrow scope and use case scenario for the developed tool. Consequently, the tool's applicability to other scenarios may be limited. However, if the thesis were to be repeated, the methodology and decisions taken could be replicated, potentially leading to different results and the emergence of a new scope for a new tool.

Furthermore, the findings related to prompt engineering are specific to the identified requirements for the tool. As discussed in Section 5.2, the suitability of different prompting techniques is specific to our use case, and prompting techniques that were deemed unsuitable may have potential usable in other contexts or settings.

5.4.4 Reliability

Reliability refers to the extent to which other researchers can replicate the study. During the preparation for the thematic analysis of the interviews, the first step involved transcribing the interviews. However, this process of converting the verbal content into written text could potentially result in the exclusion of specific emotions or nuances expressed by the participants. It is also important to note that the interviews were initially conducted in Swedish, and were therefore transcribed in Swedish. However, for the purpose of the thematic analysis, the transcriptions were later translated into English. It is possible that during this translation process, some information may have been lost or altered due to English not being our native language.

In addition to this, when employing thematic analysis, there is a risk that other researchers may not employ the same coding approach when analyzing the interviews, which also could affect the reliability of the study. It is also worth noting that we are not professional software testers, and may have overlooked important aspects of various answers when coding the transcribed interviews.

Furthermore, replicating the results of RQ2 could pose some challenges. The difficulty arises from the fact that we have not provided specific details about the input and output contents, and the prompts used in the analysis were manually crafted. Consequently, future research attempting to test the same techniques may yield different results since the prompts might not be defined in precisely the same manner, leading to varied outputs and consequently different outcomes.

Finally, it is important to acknowledge that ChatGPT has been utilized in the process of writing this Master's Thesis. However, from guidelines and instructions from both the supervisor and examiner, we have refrained from relying on ChatGPT to generate paragraphs or provide insights into our methodologies and results. Its usage has been limited to refining our own thoughts and ideas, making them more readable for you as a reader. To clarify, we would write a few sentences and then seek suggestions from ChatGPT to improve their quality. It is crucial to note that while using this tool, we remain aware of the potential inaccuracies and biases associated with LLMs. As a precaution, we have continuously proofread and verified the suggestions provided by ChatGPT to ensure the validity and overall quality of our work.

5.5 Potentials and Limitations of the Tool

The potential of the developed tool is immense. We have discovered that even complex tasks involving inputs that are not formally manufactured can still be analyzed. During one of our interviews, we posed the question, "Why do you think a tool like this doesn't exist today?". One of the participants responded by stating that it is a highly challenging problem to solve. We believe this is partly because all the possibilities that will arise with these LLMs are yet to be discovered. We have targeted a rather niche use case where automation has previously proven difficult due to the diverse formats of the inputs. However, we acknowledge that this still remains a limitation of our tool. Nevertheless, through further development and research, we have observed that it is possible to overcome this limitation with prompt engineering and LLMs.

It is worth noting that the current version of our developed tool utilizes the GPT-3.5 model. OpenAI released GPT-4 in a limited form on March 14th, 2023. GPT-4 is a similar language model that has been trained on six times the amount of data compared to GPT-3.5. According to reviews, GPT-4 shows even more significant potential and accuracy than its predecessor, enabling it to comprehend and solve even more complex tasks. If we were to implement GPT-4 into our tool, there is a strong possibility of achieving a significant increase in accuracy. However, it is important to emphasize that this is speculative, and a thorough analysis would need to be conducted to confirm this.

Furthermore, we faced difficulties in obtaining the necessary data for our analysis. This is understandable given our partnership with a consulting firm, where the available data primarily belongs to their clients. Since our tool relies on API requests, this implies that we would be transmitting sensitive information to OpenAI with limited control. Although the data is encrypted during API usage, the request for data raised concerns among multiple stakeholders. One possible solution to this issue is to utilize an alternative LLM that supports local hosting. We experimented with this approach by implementing GPT-J, an open-source LLM trained on 6 billion parameters, in contrast to GPT-3.5, which is trained on 175 billion parameters. Despite GPT-J being nearly 30 times smaller, we discovered that running the model locally requires significant computational resources. Moreover, due to its reduced size, GPT-J exhibits greater inaccuracies, which motivates our decision to opt for GPT-3.5. Nevertheless, some sources suggest the potential release of GPT-4 for enterprise use [2]. This new version aims to provide greater data control during model prompting and enable local hosting of the models. While OpenAI has yet to provide full confirmation, this possibility presents a potential solution for companies to keep their data in-house. It may also affect the financial cost since by running and hosting the model locally the cost for API usage is removed.

Another aspect is regarding one of the requirements elicited for RQ1 (see Section 4.1). This requirement states that "The tool should be capable of processing various inputs, including manual tests in natural language and automatic tests". However, it is important to note that this requirement is currently not fulfilled. Presently, the tool only supports CSV files containing requirements and test cases written in natural language as input. Therefore, the capability to process automatic tests or code-based inputs is unavailable in the current implementation. Nonetheless, we have observed that the underlying LLM utilized by the tool has the potential to support automatic tests written as code, given its extensive training on both code and natural language. Due to the constraints of time inherent to a Master's Thesis, the implementation of this functionality was not prioritized within the given timeframe.

An additional constraint of the tool has been outlined in Section 4.3. Currently, it is not optimized to process substantial volumes of data for analysis efficiently. Since the tool relies on API communication to conduct the analysis, an increase in input data prompts a corresponding increase in API requests. Consequently, the analysis process becomes considerably slower as the volume of data grows. Furthermore, it is important to consider the cost associated with each token sent through the API. This implies that as the amount of input data increases, the utilization of the tool becomes more expensive. Therefore, the tool's performance is hindered by the time-consuming analysis process and the potential financial implications of extensive data usage.

It is important to recognize that the current solution for this tool relies solely on an LLM. It is well-known that LLMs can occasionally produce inaccurate responses, as evidenced by the accuracy analysis in 4.3. Consequently, further development of the tool is necessary to ensure the generation of reliable and trustworthy results, particularly considering its tendency to generate false positives when aligning requirements and test cases. Additionally, the effectiveness of an LLM is contingent on the quality and diversity of the data on which it has been trained. Given the ex-

---

[2] `https://technews180.com/openai-announces-new-chatgpt-business-subscription-for-enterprises/`

tensive and varied training data used for LLMs, it becomes challenging to determine the underlying basis for their responses. This raises concerns regarding the potential bias inherent in LLMs. As a result, whenever the LLM provides a response, it becomes difficult to ascertain its truthfulness, as it may be influenced by the opinions and biases of the individuals reflected in the training data.

6

Conclusion

Within this Master's Thesis, we explore the topic of prompt engineering and the utilization of an LLM for a specific and complex scenario. Through the process of eliciting tool requirements, our findings indicate that the area of testing that requires the most automation lies within the system level, as the existing tools currently fall short. This inadequacy stems from the complex nature of handling diverse formats and input types in which requirements and test cases are provided. Consequently, we turned our attention to LLMs, which have demonstrated the ability to process various inputs effectively. By implementing prompt engineering techniques, we devised a solution capable of aligning requirements with test cases, addressing the challenges inherent in the task.

Based on the results, it is evident that achieving 100% accuracy in aligning requirements and tests with the current state of the tool is unattainable. To establish a reliable and trustworthy tool suitable for professional use, further development is necessary. However, considering the rapid advancements in LLM technology and its growing improvements, we view this solution as viable for successfully tackling specific and complex tasks in the near future. As the models continue to evolve, there is potential for significant advancements and enhanced capabilities, allowing for more accurate and reliable alignment of requirements and tests.

Our research has gained interest from diverse stakeholders who recognize the potential of a tool like ours. Despite the fact that the alignment may not be flawless, it offers a fresh perspective on requirements specifications and their corresponding test suites. By providing insights and suggestions on how to test specific requirements, the tool serves as an additional resource, offering a new set of eyes to enhance the testing process.

6.1 Future Work

This Master's Thesis opens up the discussion of new complex topics that may implement LLMs in order to automate labor-extensive tasks. Our results show relatively good results while only focusing on fine-tuning the prompts for the best results. Nonetheless, when it comes to the domain of aligning requirements and tests using LLMs, a challenging question remains, "How can you ensure that a requirement is fully met?". With our tool as a pilot project, we lay the foundation for expanding its capabilities and achieving greater depth. An avenue for enhancement involves incorporating additional metrics, such as coverage, which could focus on addressing the question: “To what extent is a requirement tested?”.

Future research within the field of prompt engineering could also adopt more strategies and find new, complex use cases with similar structures. For instance, this could involve the utilization of versatile tools capable of operating across all levels of software testing, including the analysis of code-based tests. LLMs could play a pivotal role in analyzing tests and categorizing testing methodologies, offering software testers approaches that have not been previously explored. Additionally, LLMs have demonstrated the ability to generate code, and with the advancement of more sophisticated models and improved prompts, we can anticipate further research and development in this area in the future.

Software testing today is still an expensive process, the utilization of advanced tools in software testing has the potential to significantly reduce costs for enterprises. By enhancing efficiency, providing valuable insights, and improving the overall quality of the software, these tools have the potential of contributing to a more cost-effective testing process, ultimately benefiting both organizations and their stakeholders.

Bibliography

[1] H. Hemmati, “How effective are code coverage criteria?” In *2015 IEEE International Conference on Software Quality, Reliability and Security*, IEEE, 2015, pp. 151–156.

[2] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel, “How well do professional developers test with code coverage visualizations? an empirical study,” in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05)*, IEEE, 2005, pp. 53–60.

[3] M. Perscheid, D. Cassou, and R. Hirschfeld, “Test quality feedback improving effectivity and efficiency of unit testing,” in *2012 10th International Conference on Creating, Connecting and Collaborating through Computing*, IEEE, 2012, pp. 60–67.

[4] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, “Tsdetect: An open source test smells detection tool,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1650–1654, ISBN: 9781450370431. DOI: `10.1145/3368089.3417921`. [Online]. Available: `https://doi.org/10.1145/3368089.3417921`.

[5] G. Sabaliauskaite, A. Loconsole, E. Engström, *et al.*, “Challenges in aligning requirements engineering and verification in a large-scale industrial context,” in *Requirements Engineering: Foundation for Software Quality: 16th International Working Conference, REFSQ 2010, Essen, Germany, June 30–July 2, 2010. Proceedings 16*, Springer, 2010, pp. 128–142.

[6] E. Bjarnason, P. Runeson, M. Borg, *et al.*, “Challenges and practices in aligning requirements with verification and validation: A case study of six companies,” *Empirical software engineering*, vol. 19, pp. 1809–1855, 2014.

[7] R. E. Kraut and L. A. Streeter, “Coordination in software development,” *Communications of the ACM*, vol. 38, no. 3, pp. 69–82, 1995.

[8] G. Wiggins, “Seven keys to effective feedback,” *Feedback*, vol. 70, no. 1, pp. 10–16, 2012.

[9] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, “Software testing techniques: A literature review,” in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016, pp. 177–182. DOI: `10.1109/ICT4M.2016.045`.

[10] S. R. Jan, S. T. U. Shah, Z. U. Johar, Y. Shah, and F. Khan, "An innovative approach to investigate various software testing techniques and strategies," *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN*, vol. 23951990, 2016.

[11] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander, "Trade-off between automated and manual software testing," *International Journal of System Assurance Engineering and Management*, vol. 2, pp. 114–125, 2011.

[12] R. Miller and C. T. Collins, "Acceptance testing," *Proc. XPUniverse*, vol. 238, 2001.

[13] J. Qadir, "Engineering Education in the Era of ChatGPT: Promise and Pitfalls of Generative AI for Education," Dec. 2022. DOI: `10.36227/techrxiv.21789434.v1`. [Online]. Available: `https://www.techrxiv.org/articles/preprint/Engineering_Education_in_the_Era_of_ChatGPT_Promise_and_Pitfalls_of_Generative_AI_for_Education/21789434`.

[14] S. S. Biswas, "Potential use of chat gpt in global warming," *Annals of biomedical engineering*, pp. 1–2, 2023.

[15] S. S. Biswas, "Role of chat gpt in public health," *Annals of Biomedical Engineering*, pp. 1–2, 2023.

[16] M. Dowling and B. Lucey, "Chatgpt for (finance) research: The bananarama conjecture," *Finance Research Letters*, vol. 53, p. 103 662, 2023.

[17] S. Biswas, *Chatgpt and the future of medical writing*, 2023.

[18] R. Dale, "Gpt-3: What's it good for?" *Natural Language Engineering*, vol. 27, no. 1, pp. 113–118, 2021.

[19] J. White, Q. Fu, S. Hays, *et al.*, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[20] H. Dang, L. Mecke, F. Lehmann, S. Goller, and D. Buschek, *How to prompt? opportunities and challenges of zero- and few-shot learning for human-ai interaction in creative applications of generative models*, 2022. arXiv: `2209.01390 [cs.HC]`.

[21] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," *arXiv preprint arXiv:2104.08786*, 2021.

[22] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[23] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–22.

[24] X. Ni and P. Li, "Unified text structuralization with instruction-tuned language models," *arXiv preprint arXiv:2303.14956*, 2023.

[25] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "Camel: Communicative agents for" mind" exploration of large scale language model society," *arXiv preprint arXiv:2303.17760*, 2023.

[26] J. Wei, X. Wang, D. Schuurmans, *et al.*, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[27] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 142–151.

[28] P. Rempel and P. Mäder, "Preventing defects: The impact of requirements traceability completeness on software quality," *IEEE Transactions on Software Engineering*, vol. 43, no. 8, pp. 777–797, 2017. DOI: `10.1109/TSE.2016.2622264`.

[29] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE transactions on software engineering*, no. 12, pp. 1278–1296, 1987.

[30] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 214–224, ISBN: 9781450336758. DOI: `10.1145/2786805.2786858`. [Online]. Available: `https://doi.org/10.1145/2786805.2786858`.

[31] E. Enoiu, G. Tukseferi, and R. Feldt, "Towards a model of testers' cognitive processes: Software testing as a problem solving approach," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020, pp. 272–279. DOI: `10.1109/QRS-C51114.2020.00053`.

[32] A. Cusmaru and E. P. Enoiu, "How aware are we of our biases in testing?" In *30th Eurostar software testing conference, 2022*, 2022.

[33] E. Knauss, "Constructive master's thesis work in industry: Guidelines for applying design science research," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, IEEE, 2021, pp. 110–121. DOI: `10.1109/ICSE-SEET52601.2021.00021`.

[34] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[35] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006. DOI: `10.1191/1478088706qp063oa`. eprint: `https://www.tandfonline.com/doi/pdf/10.1191/1478088706qp063oa`. [Online]. Available: `https://www.tandfonline.com/doi/abs/10.1191/1478088706qp063oa`.

[36] C. Rivas, "Coding and analysing qualitative data," *Researching society and culture*, vol. 3, no. 2012, pp. 367–392, 2012.

[37] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[38] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, pp. 131–164, 2009.

A

Interview Guide

Settings

- Location.
  - In-person or online.
- Apparatus
  - Laptop for manuscript and transcription.
  - Audio recording device (for more detailed transcription afterward).

Start

- A short introduction to the purpose of the thesis (creation of a tool that's aimed to support software testers).
- A short explanation of why we are conducting this interview (to identify relevant requirements for the tool).
- Explain the interview contract:
  - The participant will remain anonymous.
  - The interview will be recorded.
  - The participant is free to cancel the interview at any time.
  - Verbal confirmation of the agreement of the contract.

Intro
(Here's where the actual interview is initiated and the recording starts, the purpose of these questions is to validate whether the interview participant is a valid subject, as well as to find out about their experience.)

- What is your role at your company? (For our case TestScouts)
  - For how long have you been working in this role?
  - (In case the subject hasn't worked as a software tester ask the following

question): Do you have any experience with software testing?

Test Process
(Questions regarding common practices, common mistakes made by software testers, common challenges within software testing, as well as identifying important quality attributes.)

- Do test suites usually exist in the projects you join or do you help build them?
    - If or when there is a test suite, what is the first thing you want to find out about it?
        - Any specific metric? For example, code coverage, size, mutation score.
- With that specified (or not), are you looking for any red flags in the test suite? For example, any obvious errors, bad practices, etc.
- How do you/your team usually work to improve the test suite?
    - Do you focus more on improving existing tests or is there more emphasis on what is not yet tested?
    - Do you use any tools to facilitate this process (i.e., to improve the test suite)?
        - If so, which ones?
    - When you feel that you have improved the test suite, how do you validate or measure whether you have made the test suite better or not?
        - Do you use any specific tools to measure this?
    - What characterizes a good test suite for you?
        - (If they don't answer with a quality attribute) Any specific quality attribute that you prioritize over others?

The tool
(Questions regarding more technical aspects; the most relevant output for a tool with this purpose, is the subject aware if such a tool already exists, in what time aspect the subject would like to receive the said feedback, the environment upon which the tool should be built, etc.)

- We want to create a tool that can help the tester write better tests, and we plan to do this by providing some form of feedback.
    - What feedback do you feel would help you the most?
        - Do you currently use any tool to receive this type of feedback?
        - Do you know if there is any tool available today that can provide you

with this feedback?

    - If not, why do you think such a tool does not exist?

- With the feedback you just mentioned, when would you prefer to receive it? For example, before you start with test development, in real-time, or afterward to analyze the changes made?

- When working with a test suite, is there anything that you usually miss that you would appreciate being reminded of?

- In what environment do you think the tool would best suit? For example, as an executable script, a plugin for some editor/IDE, or something else?

- Is there any functionality/feature that you would like to see in this tool that we may not have thought of?

- Since we will be working on this project throughout the spring, could we use you as a sounding board in case of any questions, new ideas, or thoughts about the tool?

- Do you have anything else you would like to add, about anything really? Otherwise, thank you very much for your help!