ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

**SCUOLA DI SCIENZE**
**Corso di Laurea in Informatica**

# Analyzing The Use Of Large Language Models In eXtreme Programming Agile Practices

**Relatore:**
**Chiar.mo Prof.**
**Paolo Ciancarini**

**Presentata da:**
**Andrea Largura**

**Sessione III**
**Anno Accademico 2022/2023**

*Alla mia famiglia,*
*che mi ha permesso di intraprendere questo viaggio.*
*Alla mia fidanzata,*
*che lo ha reso indimenticabile.*

# Abstract

In this dissertation we investigate the utilization of Large Language Models (LLMs) to enhance the effectiveness of Extreme Programming (XP) practices in agile software development. The study delves into various XP practices through a mixed-methods approach combining quantitative and qualitative analysis. The overarching aim is to discern how LLMs can augment human developers' capabilities within agile practices, examining aspects such as efficiency, collaboration dynamics, code quality, and overall productivity. Our study reveals that the practices that tend to work better with LLMs include those that involve repetitive tasks and require extensive code generation or manipulation, such as Test-Driven Development (TDD) and collaborative programming scenarios. Despite the need for human validation, LLMs can significantly enhance productivity in these contexts. Conversely, practices relying heavily on human judgment, such as user story and use case evaluation, may not benefit as much from LLM integration. We conclude with recommendations for future research such as improving LLMs prompts, and addressing security issues.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In recent years, the field of Natural Language Processing (NLP) has developed a significant transformation with the development of Large Language Models (LLMs) [48]. These models, built upon advanced deep learning architectures, have demonstrated remarkable capabilities in understanding, generating, and manipulating both human language and formal notations. Led by breakthroughs such as OpenAI's GPT (Generative Pre-trained Transformer), LLMs have achieved unprecedented levels of performance across a wide range of tasks, including code generation and elaboration.

## 1.1 Context and Relevance: Role of LLMs in Software Engineering

The impact of LLMs extends beyond traditional NLP domains and into the realm of software engineering. Software development is inherently a linguistic activity, involving the creation and manipulation of code artifacts, documentation, and communication among team members. LLMs offer unique opportunities to improve and simplify various aspects of the software development lifecycle by taking advantage of their language understanding and generation capabilities.
The integration of LLMs into software development practices holds promise

for enhancing productivity, code quality, and collaboration among developers [19], [32]. By harnessing the power of LLMs, developers can automate repetitive tasks, generate code snippets, provide natural language explanations, and assist in debugging and testing activities. Moreover, LLMs can facilitate knowledge sharing, code reviews, and documentation efforts, leading to more efficient and transparent development processes. We aim to provide a comprehensive evaluation that not only highlights the advantages but also sheds light on areas where improvements or alternative strategies may be necessary to optimize the effectiveness of LLMs in agile software development contexts [13].

## 1.2 Research Questions

In this context, our research endeavors to assess the feasibility and effectiveness of integrating LLMs to assist human developers within the context of Extreme Programming practices. As a result, the following research questions arise:

**RQ1** How can LLMs be effectively integrated into agile software development practices to improve productivity, code quality, and collaboration among developers?

**RQ2** What are the potential challenges and limitations associated with the adoption of LLMs in agile software engineering contexts, and how can they be mitigated?

## 1.3 Methodology Employed

To investigate these research questions, a mixed-methods approach will be employed, combining quantitative analysis of software development repositories with qualitative analysis based on established criteria, as well as surveys conducted with software developers. This combined methodology ensures a

comprehensive exploration of both objective metrics and subjective experiences related to the integration of LLMs in software development practices, offering valuable insights for future research.
We will conduct a thorough quantitative analysis of software development repositories to examine the usage patterns of LLMs in some projects. This quantitative analysis will provide valuable insights into the extent to which LLMs are utilized, the types of tasks they are employed for, and their impact on productivity and code quality.
In addition to quantitative analysis, we will conduct qualitative analysis. Initially, we will establish criteria developed by our team or inspired by existing research for evaluating various aspects of the data. Additionally, we will engage developers through surveys to capture their subjective assessments and insights to introduce an additional criterion for evaluation.

## 1.4 Structure of the Thesis

In this thesis, Chapter 2 presents and provides an evaluation of some Extreme Programming practices when integrated with LLMs, offering a distinct analysis for each practice. Chapter 3 offers a comprehensive discussion while Chapter 4 summarizes the study's findings.

# Chapter 2

# Extreme Programming

In this chapter, we delve into the realm of eXtreme Programming (XP), an agile software development methodology known for its emphasis on best practices for flexibility, collaboration, and responsiveness to change [8]. An image illustrating the XP practices, as envisioned by one of its co-creators, Ron Jeffries, can be seen in Figure 2.1 [22].

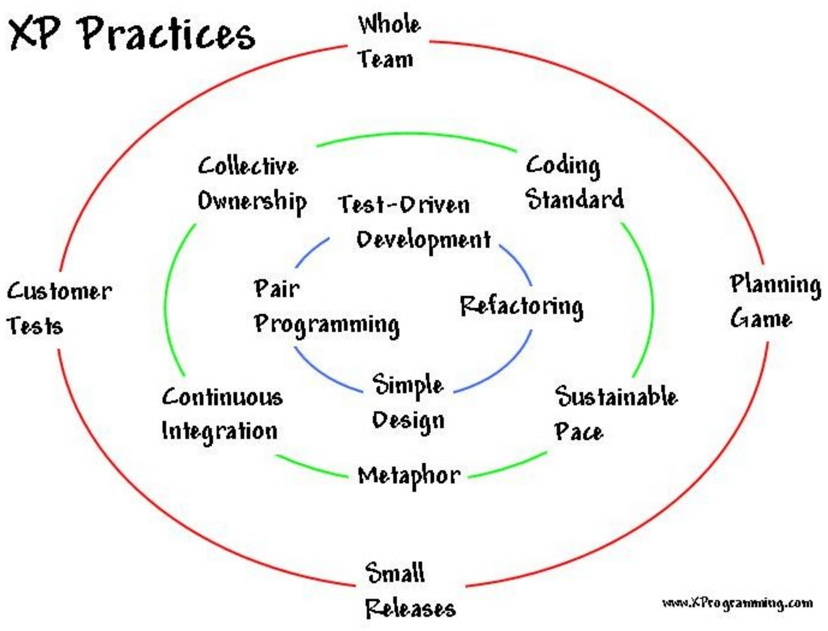


Figure 2.1: XP Practices from [22]

While XP encompasses a wide array of practices, our attention is directed towards key methodologies within this framework. Specifically, we emphasize practices that are not only fundamental but also highly practical. These practices include Test-Driven Development (TDD), Pair Programming, Code Refactoring, Code Quality Assurance, Continuous Integration (CI), and the formulation and implementation of Use Cases and User Stories, along with the pivotal role of Unit Tests.
The rationale behind this selective analysis arises from the significance and impact of these practices on the overall agile software development process. TDD, for instance, revolutionizes the approach to writing code by prioritizing test creation before implementation, ensuring robustness and functionality from the outset. Pair Programming fosters collaboration and knowledge sharing, enhancing code quality and reducing errors. Similarly, Code Refactoring and Quality Assurance practices ensure that the codebase remains maintainable, scalable, and adheres to established standards. Continuous Integration plays a vital role in ensuring that code changes are continuously integrated into the main codebase, facilitating early detection of conflicts and errors. Moreover, the formulation and refinement of Use Cases and User Stories provide a structured approach to understanding and addressing user requirements, guiding development efforts towards delivering value.
By leveraging the capabilities of LLMs, we aim to investigate their potential applications in the context of agile software engineering, with a particular emphasis on software testing and quality assurance. Through empirical analysis and experimentation, we attempt to evaluate the efficacy of LLMs in supporting XP principles and facilitating agile development processes.

## 2.1 Test Driven Development

Test Driven Development (TDD) is a development methodology aiming to improve software quality [34]. As the name suggests, the tests drive the development process, as first the tests and then the software are written. If the software passes all test cases we can now refactor the code then create a new test and repeat the cycle. In his book 'Test Driven Development: By Example', Kent Beck, to whom TDD is attributed, outlines two fundamental rules for using TDD: 'write a failing automated test before you write any code' and 'remove duplication' [9]. Thanks to those two rules, we can gain an understanding of how TDD works. The efficiency of TDD for improving software source code generation using the LLMs has been recently studied in [29], however, we intend to perform a separate and indipendent investigation.

### 2.1.1 The Tasks

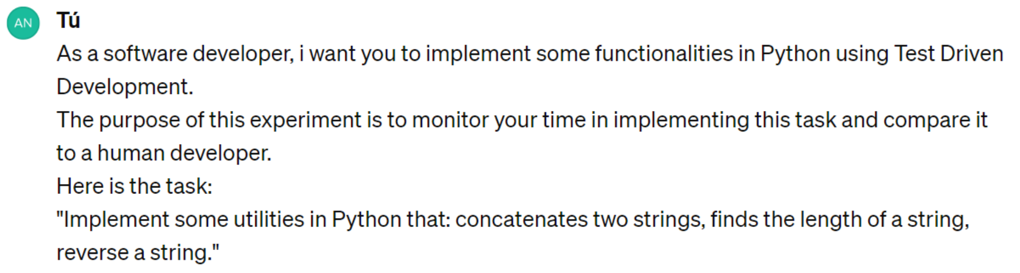
In order to compare how helpful can LLMs be during the process of writing tests and code using a TDD approach, the same three tasks have been assigned to both human and AI for comparing the results in terms of time. The chosen tasks span a spectrum of difficulty, ranging from a fundamental implementation of some string manipulation functionality to the development of a basic authentication system.
The tasks assigned consisted of developing a string manipulation utility, a user registration scenario, and an authentication system. These scenarios cover basic string manipulation, user interaction, and security aspects, providing a varied set of challenges to assess the performance of both human and AI developers in different software development tasks. It is crucial to note that there is a tight correlation between the second and the third task; the reason for this will be explained later in Section 2.1.4.

### 2.1.2 Human-driven TDD

A diverse group of 8 skilled programmers with varying levels of experience participated in the experiment. Each participant was provided with detailed task descriptions. The participants were asked to independently implementing the assigned scenarios using Python on their preferred development environments. All the human participants completed successfully the assigned tasks, it is crucial to clarify that the recorded completion times were measured from the moment each individual understood the task and started writing tests and code. This approach ensured a consistent evaluation of their performance.

### 2.1.3 AI-driven TDD

In conducting our evaluation, the model employed for generating responses to the given prompts was GPT-3.5. Below, in Figure 2.2, we show the prompt utilized to evaluate the performance of AI systems in TDD (in the case of the string manipulation utility): To better understand how the



**Tú**

As a software developer, i want you to implement some functionalities in Python using Test Driven Development.
The purpose of this experiment is to monitor your time in implementing this task and compare it to a human developer.
Here is the task:
"Implement some utilities in Python that: concatenates two strings, finds the length of a string, reverse a string."

Figure 2.2: TDD Prompt

evaluation of the AI was done, it is crucial to note that all the prompts were phrased as natural language descriptions or instructions specifying the task to be performed. Our approach involved initially presenting contextual information related to the TDD experiment to GPT-3.5. Subsequently, precise sequences of instructions that specify the desired outcome were constructed, followed by the details of the task. This format aimed to guide the AI in generating responses useful for the desired result.
The responses to all the tasks were generated within ten minutes, and it exhibited impeccable accuracy if it wasn't for a little misunderstanding in the way the time was going to be recorded. GPT-3.5 placed the monitoring of the time in the code snippet provided, while the scope of the experiment was to monitor from the moment the prompt was given to the moment a correct

output was received, since the intention was to monitor the AI's processing time comprehensively, as a result, the prompt was given back to ChatGPT removing the part about the purpose. As a consequence, the result changed from this:

```
def test_concatenate_strings(self):
        start_time = time.time()
        result = concatenate_strings("Hello", "World")
        end_time = time.time()
        print("Time taken to concatenate strings:", end_time -
        start_time)
        self.assertEqual(result, "HelloWorld")
```

To this:

```
def test_concatenate_strings(self):
        # Test if concatenate_strings function concatenates tw
        o strings correctly
        self.assertEqual(concatenate_strings("hello", "world")
        , "helloworld")
```

### 2.1.4 Evaluation

After recording the results of the experiment it is possible to provide valuable insights and a deeper understanding of the outcomes achieved during the experimental process. Below in Table 2.1 we show a graphical representation of the average time taken by human participants and AI to complete each task.

In the experiment, subjects executed the following steps: read the task specifications, write a test case, implement source code that pass the test and repeat these steps until the task is completed.

Table 2.1: Average Implementation Time in TDD

| Method | Task 1 | Task 2 | Task 3 |
|---|---|---|---|
| Human | 6m21s | 13m39s | 4m21s |
| AI | 36s | 2m27s | 2m49s |

#### 2.1.4.1 Evaluation of Implementation Time

As shown in Table 2.1, overall the data suggests that the AI consistently performed faster than human developers across all tasks. The time for the implementation step in TDD was reduced by 95% for easier tasks like Task 1 and by 58% for harder tasks such as Task 2 and Task 3. The results align with findings from [29], where a 94% decrease in implementation time for easier tasks and a 56% reduction for more challenging ones was reported. This congruence between our findings and their research underscores the potential of AI to simplify development processes in TDD.
In Task 3, we observed only a 34% reduction in implementation time. The measure decreased compared to it's very similar task, Task 2 being completed 82% faster compared to the corresponding human-developed implementation, this can be attributed to a congruency with the challenges encountered in the second task. Such similarity enabled human developers to adapt more effectively, resulting in improved performance despite the complexities presented in Task 3.

#### 2.1.4.2 Evaluation of the Source Code

During the experiment, AI's generated source code was in most of the cases correct and accurate. However, in certain instances, the produced code did not pass the testing. Generating incorrect source code increases developer effort and development time [29]. A more comprehensive study is warranted, considering not only correctness but also robustness in the face of varying testing scenarios.

## 2.2 Pair Programming

"Pair programming is a software development technique where two programmers work together at one workstation" [47]. In this collaborative approach the one who uses mouse and keyboard and writes code is called "driver" and he has a short-term perspective, while the other one called "observer" or "navigator" continuously observes the work of the driver, suggesting improvements, helping catching errors, he notes the missing tests and possible code refactoring during development, he focuses on the long term. These roles can be traded in order to improve software quality.
It is imperative to conduct a thorough evaluation of human-AI pair programming, given the limited understanding of its potential benefits and drawbacks, as highlighted in [27].
The experiment involved two computer science students collaborating, while in the human-AI pair programming scenario, one student collaborated with the AI assistant. Their primary objective was to collaboratively develop, implement and subsequently test some functionalities for React components.

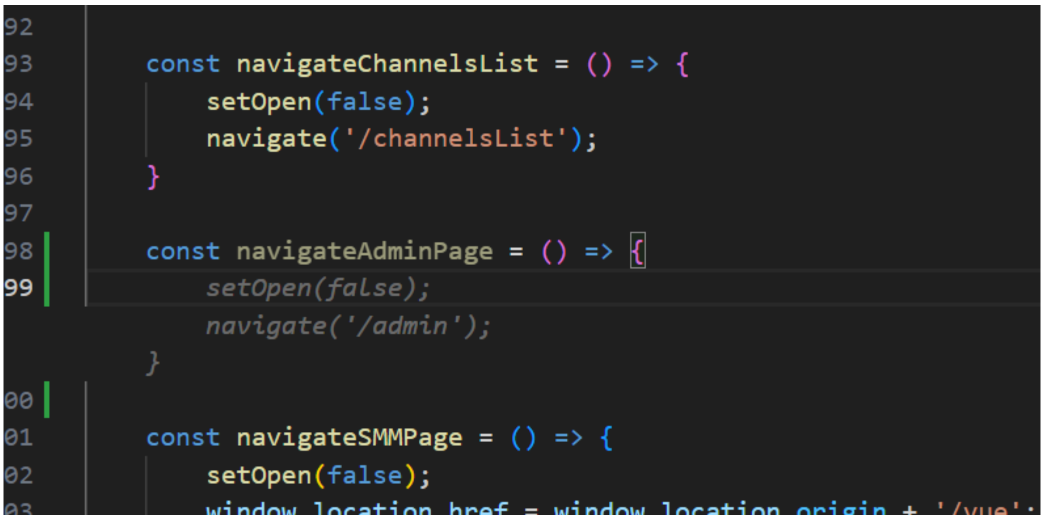
### 2.2.1 Human-Human Pair Programming

Pair programming with a human partner offers numerous advantages in a collaborative software development setting, the constant exchange of knowledge between team members stimulates a rich learning environment, allowing developers to share techniques and best practices. Furthermore having two sets of eyes on the code serves as an effective mechanism for error prevention, this continuous feedback not only reduces the expectation of bugs but also improves the overall quality of the code produced.

### 2.2.2 Human-AI Pair Programming

"AI pair programming is a technique that involves the use of AI to assist a developer in writing code" [7]. This experiment investigates the effectiveness of pair programming with an AI-powered code completion assistant called

Tabnine[1], an alternative to GitHub Copilot.
It's essential to note that unlike traditional prompts provided to AI models, Tabnine operates by offering coding suggestions and completions without the necessity of explicit prompts. As depicted in Figure 2.3, these suggestions are presented to the programmer for consideration. It's worth mentioning that Tabnine provides multiple completion options, allowing the programmer to choose among them. However, it's important to acknowledge that while Tabnine's completions can be valuable, they may not always align with the programmer's intentions, leading to potential inaccuracies or inconsistencies in the suggested code.


```

    const navigateChannelsList = () => {
        setOpen(false);
        navigate('/channelsList');
    }

    const navigateAdminPage = () => {
        setOpen(false);
        navigate('/admin');
    }

    const navigateSMMPage = () => {
        setOpen(false);
```

Figure 2.3: Pair Programming Prompt
(The code in gray is the code suggested by Tabnine)

Our results suggests that AI pair programming can result in generating additional lines of code within a shorter timeframe compared to human pair-programming. However the price for this is a regression in code quality, moreover it can lead to inaccuracy and cause more errors and tests to fail. This findings align quite well with a GitHub Copilot study, which states: "although programming with Copilot helps generate more lines of code than

[1]https://www.tabnine.com

human pair-programming in the same period of time, the quality of code generated by Copilot appears to be lower" [20]. In any case, it is especially useful for repetitive tasks, so developers can focus on more complicated tasks and solving problems [7].

### 2.2.3 Evaluation

We observed distinctive patterns in productivity and code quality. The evaluation comprehend multiple aspects, including efficiency, collaboration dynamics, and the reliability of the generated code. We encountered a 97% accuracy for the human-human part and an 88% for the human-AI one.

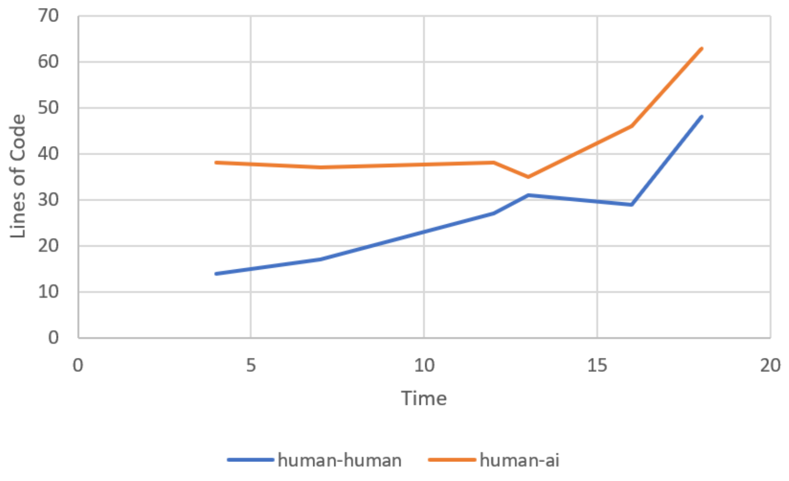


Figure 2.4: Lines of Code over Time

#### 2.2.3.1 Evaluation of Efficiency

In terms of efficiency the human-AI pair programming exhibited a notable advantage, thanks to the code generation and autocomplete capabilities the coding process accelerated, resulting in a higher volume of code produced as shown in Figure 2.4. This increased productivity is indicative of the effectiveness of human-AI collaboration in software development.

#### 2.2.3.2 Evaluation of Collaboration Dynamics

Collaboration dynamics in human-human pair programming were characterized by an effective communication, the collaborative approach between driver and navigator allowed a shared problem-solving, facilitating the code development. On the other side the human-AI pair programming the human developer collaborated with Tabnine mainly through code suggestions and autocompletions. The observed increase in the number of failed tests in human-AI pair programming scenarios, despite the higher productivity in terms of lines of code produced, may likely be attributed to two primary factors: firstly, as indicated in Figure 2.4, the dynamics of a more individualized coding experience could play a significant role and secondly, the limitations inherent in AI paired programming, due to its training data limitations [11].

#### 2.2.3.3 Evaluation of Code Quality

The code produced through human-human pair programming presented a balance between quality and quantity, as evidenced by a 97% accuracy rate in Section 2.2.3. With two developers working together, the code is continuously reviewed and refined, resulting in a higher standard of quality [2]. Contrastingly, the code generated with Tabnine showed a propensity for higher inaccuracy and lower quality. The rapid code generation and autocomplete features contributed to increased output, nevertheless it's important to be cautious about maintaining accuracy and quality in the code generated by human-AI collaboration.

## 2.3 Code Refactoring

Refactoring is a systematic process to improve internal code by making many small changes without altering the code's external behaviour. Or according to the definition by Fowler et al., "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [16].

There are several reasons why code refactoring is crucial in software engineering, such as keeping your code clean and readable, improve the performance of your application by removing unnecessary lines of code, make bugs easier to find. Moreover not doing so can have bad consequences for example the technical debt may increase and in the future not having a clean code may make it harder to implement new features due to code smells.

### 2.3.1 Human and AI Refactoring: Pro and Cons

"Most code refactoring tasks are performed manually by developers based on experience and best practices" [30]. This manual process requires a deep understanding of the code, including its underlying logic and structure. The process of refactoring often consumes a significant amount of time for developers, time that could otherwise be allocated to implementing new features or functionality. Yet, it is necessary for long-term maintainability, as software systems need to go under modifications, improvements and enhancements in order to cope with evolving requirements [41].
On the other side AI refactoring often demonstrates superior speed, particularly when dealing with straightforward refactoring tasks. However, its efficiency may diminish when faced with more complex scenarios or technical intricacies.
One of the primary benefits of AI refactoring is its ability to meticulously comment all the code that have been refactored. This remarkable documentation ensures that every adjustment and improvement made to the codebase is clearly annotated, providing a detailed record of the changes implemented during the refactoring process.

### 2.3.2 Contextual Code Refactoring

The evaluation of both human and AI refactoring experiments was conducted by refactoring components of a React website, specifically a social media platform, to assess whether all functionalities remained intact and

operational. This involved a systematic revision of the code to improve its structure, readability, and maintainability while ensuring that no existing features were compromised or rendered dysfunctional due to the refactoring process.
It is necessary to specify that the developer tasked with refactoring is the original author of the code being optimized. This ensures that the developer not only comprehends how the code operates but also recognizes areas for potential enhancement. On the AI-side, the refactoring was done by GPT-3.5, with a simple prompt:

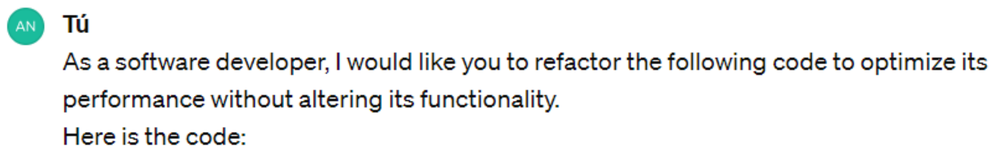

AN Tú
As a software developer, I would like you to refactor the following code to optimize its performance without altering its functionality.
Here is the code:

Figure 2.5: Refactoring Prompt
(The corresponding code can be found in the appendix)

### 2.3.3 Evaluation

Below in Table 2.2 is a comparison of code refactoring performance between AI and human developers. The table presents data on the total number of lines reduced, the percentage reduction, the average rate of lines refactored per second and per minute, and the average time taken for the refactoring process.

Following, Figure 2.6 provides a overview of the code refactoring activities performed by GPT-3.5. It presents various metrics and statistics related to code changes aimed at improving code quality, readability, and maintainability.
It's necessary to note that the AI was unable to refactor a component of approximately 500 lines, a significant portion compared to the total number of lines which was 827, hence those lines were not included in Table 2.2 and Figure 2.6 calculations. Additionally, it's worth mentioning that the human

Table 2.2: Comparison of Code Refactoring Performance

| | AI | Human |
|---|---|---|
| Total lines reduced | 137 | 57 |
| Percentage of lines reduced | 16.5% | 6.89% |
| Lines per second | 0.706l/s | 0.095l/s |
| Lines per minute | 42.36l/m | 5.7l/m |
| Average Time per Refactoring | 34s | 1m40s |

developer assigned a predetermined time of 10 minutes for the refactoring task, which is roughly three times the duration taken by the AI for all the refactoring activities. After the designed time, the human developer ceased coding.

#### 2.3.3.1 Evaluation of Efficiency

The AI demonstrated superior performance in terms of refactoring efficiency, as evidenced by the higher number of lines reduced per unit of time compared to the human counterpart. However, it's important to note that a significant portion of the refactoring process involved small code segments. For instance, functions displayed in just three lines, often comprising a single instruction, were transformed into inline functions, while the human developer did not optimize these functions and left them in their original three-line format. This restructuring resulted in a reduction of two lines of code for each function of this type. Furthermore, it's noteworthy that the reduction in lines of code was also attributed to the removal of commented logging statements for debug and multiple empty lines as seen in Figure 2.6. While these operations contributed to the overall reduction in lines of code, it also highlights the AI's ability to automate repetitive and straightforward refactoring tasks efficiently.

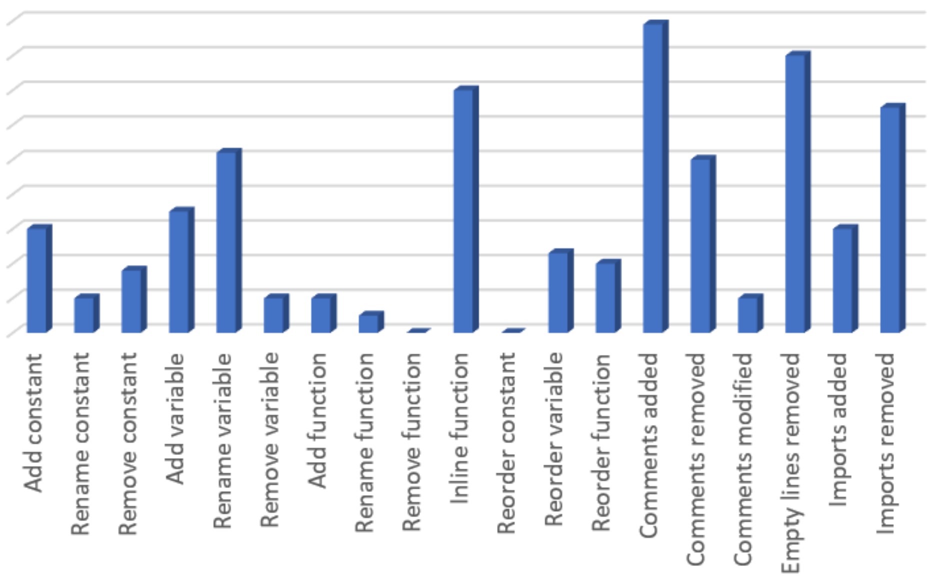


Figure 2.6: AI Refactoring Summary

#### 2.3.3.2 Evaluation of Performance

The performance of AI in refactoring small portions of code is commendable. For concise functions or code snippets, the AI demonstrated efficiency, swiftly identifying and implementing optimizations. However, as the volume of code increases, the reliability of the AI decreases in comparison to human intervention. This suggests that AI refactoring may not be as useful when handling big projects with several lines of code. As highlighted in [42], AI remains far from securely modify existing code without human supervision. One approach to enhance efficiency when dealing with large codebases is to divide the code into smaller, more manageable sections for AI refactoring. After refactoring each section individually, the code can be merged back together. However, this process might be time-consuming, and manual refactoring could potentially be more efficient. Additionally, providing the AI with a few-shot example before the refactoring process, as explored in [39], can improve the quality of the outcomes.

## 2.4 Code Quality

”Code quality refers to the overall standard and excellence of a software program’s source code, it encompasses various aspects that contribute to the code’s readability, maintainability, efficiency, and robustness” [23].
However code quality is a concept that often defies precise definition. Its interpretation can vary significantly among different teams, influenced by contextual factors, project requirements, and individual developer perspectives [10], [31].

### 2.4.1 Evaluation Metrics

With the groundwork laid in previous sections, we now turn our attention to a detailed analysis of code quality. While our earlier investigations delved into various aspects of test driven development, pair programming and code refactoring, a comprehensive assessment of code quality was deferred until now. Taking advantage of the data collected from the previous examinations, we will now examine the quality of the code produced by LLMs.
Before delving into the analysis of results, it is imperative to establish the criteria utilized for assessing the quality of code. This criteria includes a spectrum of metrics designed to evaluate various dimensions of code quality, each playing a vital role in shaping the overall efficacy and reliability of software systems. Additionally, we will utilize SonarQube[2], a widely-used platform for continuous inspection of code quality. SonarQube offers a range of static code analysis tools that evaluate code against a set of predefined rules to detect issues such as bugs, vulnerabilities, and code smells. By leveraging both the criteria explained before and SonarQube, we aim to gain a complete understanding of the code quality, considering both qualitative metrics and the technical insights offered by SonarQube’s static code analysis.

In the succeeding passages, we will provide an overview of the evaluation

[2]https://www.sonarsource.com/

metrics adopted, encompassing dimensions such as reliability, maintainability, portability, and reusability. By exploring these qualitative metrics and seizing their importance throughout the software development lifecycle, we establish the foundation for a thorough evaluation of the codebase's quality.

**Reliability**
Reliability is the probability that a system will operate without failure over a specific period of time. It measures the stability of the software.

**Maintainability**
Maintainability measures how easily software can be maintained. There are several metrics that defines maintainability, such as size, structure and complexity of the codebase.

**Portability**
Portability measures how usable the same software would be in different environments.

**Reusability**
Reusability measures wether existing piece of codes such as functions can be used again in the codebase, in other words if the code is modular.

### 2.4.2 Qualitative Evaluation

#### 2.4.2.1 Evaluation of Reliability

In terms of reliability, it's notable that code generated or refactored by AI can occasionally crash after testing certain functionalities. Additionally, despite being less common, it may crash directly after execution, often due to static errors.
Figure 2.7 illustrates the percentage of prompts required before a reliable code solution was received from the AI. The analysis reveals that in the vast

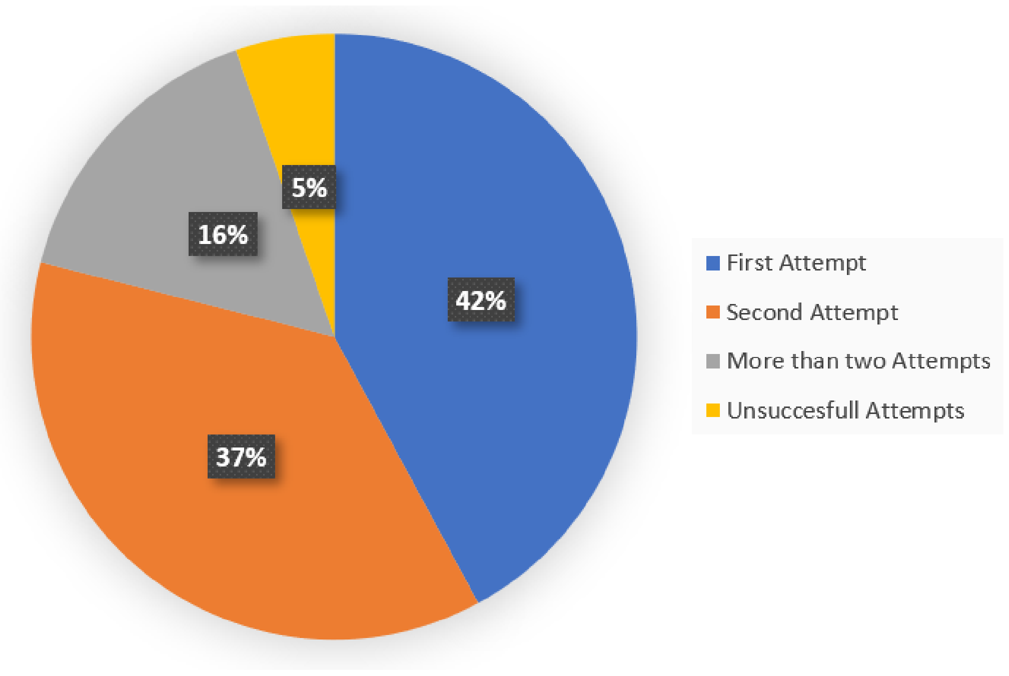


Figure 2.7: AI Code Reliability Analysis

majority of instances, approximately 95% of the time, the AI effectively delivers a reliable code solution. This suggests an adequate level of performance and reliability in generating accurate code outputs. However, there exists a distinct subset, constituting about 5% of cases, where the AI's performance falls short in providing a satisfactory solution. This discrepancy could be attributed to various factors, including the intricacy or extensive nature of the code under consideration.

#### 2.4.2.2 Evaluation of Maintainability

When it comes to maintainability, the AI demonstrates a commitment to adhering to the best practices in software development. It excels in breaking down code into manageable and modular components. By embracing these principles, the AI not only enhances the readability and understandability of the codebase but also lays a solid foundation for future maintenance and scalability efforts. While advancements have been made in evaluating code generated by LLMs, it's important to note that this field is still in its early stages of development and requires more extensive research and exploration [45].

#### 2.4.2.3 Evaluation of Portability

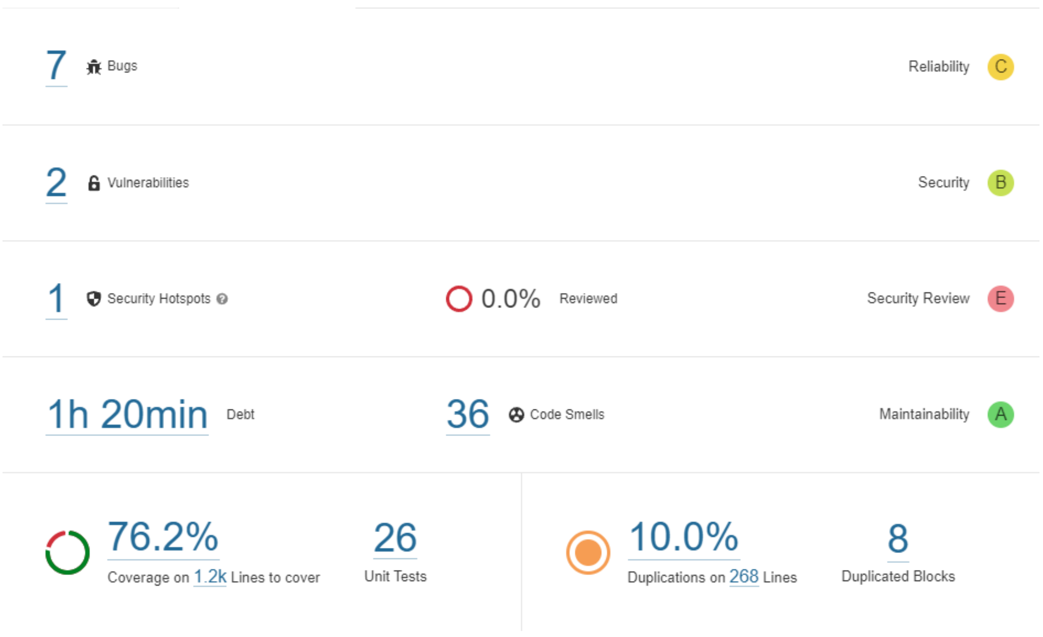
In terms of portability the solutions generated by LLMs were able to maintain functionality and performance across various platforms and devices. The AI's proficiency in this regard becomes particularly evident when tasked with refactoring or editing React components, which often involve considerations for mobile rendering and cross-platform compatibility.

#### 2.4.2.4 Evaluation of Reusability

When assessing the reusability of AI-generated code, it's essential to evaluate the modularity, coherence, and flexibility of the codebase. The AI's approach to code generation often involves breaking down complex tasks into smaller and reusable code structures. The AI's output may consist of well-defined functions, classes, or modules that encapsulate specific functionality or logic which can be easily incorporated into other projects or extended to accommodate new requirements without necessitating significant modifications to the existing codebase.

### 2.4.3 SonarQube Evaluation

The SonarQube evaluation utilized the standard SonarQube rules available within the Community Edition of SonarQube. These rules are designed to provide comprehensive insights into various aspects of code quality, including reliability, security, maintainability, and test coverage. The evaluation reveals several areas for improvement in our codebase (Figure 2.8).



7 Bugs — Reliability C

2 Vulnerabilities — Security B

1 Security Hotspots — 0.0% Reviewed — Security Review E

1h 20min Debt — 36 Code Smells — Maintainability A

76.2% Coverage on 1.2k Lines to cover — 26 Unit Tests

10.0% Duplications on 268 Lines — 8 Duplicated Blocks

Figure 2.8: SonarQube Code Quality Analysis

#### 2.4.3.1 Evaluation of Reliability

In terms of reliability, our codebase exhibits 7 bugs, classified as class C according to the SonarQube analysis. In SonarQube's classification system[3], class C indicates the presence of at least 1 Major Bug. These findings highlight areas where our software may be susceptible to significant quality flaws that could highly impact the developer's productivity, such as uncovered pieces of code, duplicated blocks, or unused parameters. The reason SonarQube reliability evaluation revealed a class C rating, is primarily due to a major bug, wherein a React hook was not properly called. However, it's worth noting that all other identified bugs were categorized as minor issues.

#### 2.4.3.2 Evaluation of Security

Regarding security, the SonarQube evaluation uncovered two vulnerabilities: one related to authentication, specifying that admin authentication

[3]https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/

should be restricted to specific IP addresses, and another concerning file upload, which should ideally be restricted. Despite these findings, both vulnerabilities were classified as minor issues, contributing to the class B rating. However, it's noteworthy to mention that these security issues were not reviewed nor fixed, resulting in a class E designation for the security review. Class E indicates that less than 30% of Security Hotspots were reviewed, reflecting a need for further attention to security vulnerabilities.

#### 2.4.3.3 Evaluation of Maintainability

In terms of maintainability, SonarQube flagged 36 code smells, resulting in a technical debt of 1 hour and 20 minutes. Despite this, the maintainability aspect was classified as class A. This classification was attributed to the technical debt being lower than 5% of the time that has already gone into the application, as determined by SonarQube. This result is particularly noteworthy and indicates a relatively high level of maintainability achieved with AI assistance.

#### 2.4.3.4 Evaluation of Duplications

Regarding duplication, SonarQube detected the presence of 8 duplicated blocks within the codebase. These blocks amounted to a total of 268 duplicated lines. When considering the entirety of the codebase (excluding comments), this duplication constituted approximately 10% of the total lines. The presence of duplicated lines, particularly when generated with the assistance of AI, raises questions about the effectiveness of the AI model in avoiding redundancy and promoting code modularity.

#### 2.4.3.5 Evaluation of Test Coverage

While in our experiment, TDD sessions resulted in full test coverage, during pair programming and refactoring sessions instead, the test coverage was reduced, consequently lowering the overall percentage of code covered by tests to only 76.2%. However, it's crucial to highlight the remarkable ability of AI in generating tests. Despite the challenges posed by dynamically generated code, AI showed promise in creating tests.

### 2.4.4 Overall Evaluation

We can infer that the combined efforts of human developers and AI assistance in code generation led to a relatively stable codebase, as indicated by the low severity of identified issues across various dimensions such as reliability, security, and maintainability. Despite the presence of some bugs, vulnerabilities, and code smells, the overall impact on the system's quality appears to be minimal, especially considering the classification of these issues by SonarQube.

The qualitative evaluation, along with SonarQube analysis, yielded congruent findings across various dimensions, particularly in reliability and maintainability. However, divergent results were observed in terms of reusability, notably regarding the presence of duplicated lines identified through SonarQube analysis. This discrepancy in reusability underscores the importance of utilizing multiple evaluation techniques for comprehensive insights into software quality.

Employing multiple evaluation techniques is crucial as each method offers unique insights into software quality. By combining qualitative assessments with automated analysis tools like SonarQube, we gain a more comprehensive understanding of the strengths and weaknesses of the software. This approach enables cross-validation, confirmation of findings, and ensures that no critical aspects of quality are overlooked, ultimately leading to more informed decisions and higher-quality software products.

## 2.5 Continuous Integration

"Continuous Integration (CI) is a software development practice where team members regularly integrate their code changes into a shared reposi-

tory" [5], often occurring multiple times throughout the day [38].
The advantages of this practice are several, such as easier and quicker error detecting, faster software testing and of course it helps deliver updates faster and more frequently. There are historical reasons for the implementation of CI in software development, that reason is that in the past, developers often operated in isolation, working independently for an extended period of time before merging their changes, this approach led to challenges in merging all the code produced by every team member, consuming considerable time and effort. Additionally, this allowed bugs to accumulate for a long time without correction, retarding the delivery of updates to customers [3], [14], [25].

### 2.5.1 Experiment Setup

For this experiment, we utilized Visual Studio Code as our Integrated Development Environment (IDE), where we configured GitHub Actions[4] as our CI platform, coupled with Tabnine as a pair programming assistant. To set up the CI pipeline, we defined workflows using YAML files stored within the repository. These workflow files specify the sequence of steps to be executed automatically in response to specific events, such as code pushes or pull requests. Within the workflow files, we configured individual jobs to perform specific tasks necessary for our CI process. These tasks included building the code and running tests. Each job was tailored to execute a particular set of tasks efficiently. The pipeline visualized in Figure 2.9 outlines the continuous integration process described. The experiment was conducted by a single individual, who evaluated their performance in CI both with and without the assistance of AI.

### 2.5.2 Error Detection and Correction

The application of AI in error detection and correction demonstrated remarkable efficacy, particularly when subjected to a process of incremental

[4]https://github.com/features/actions

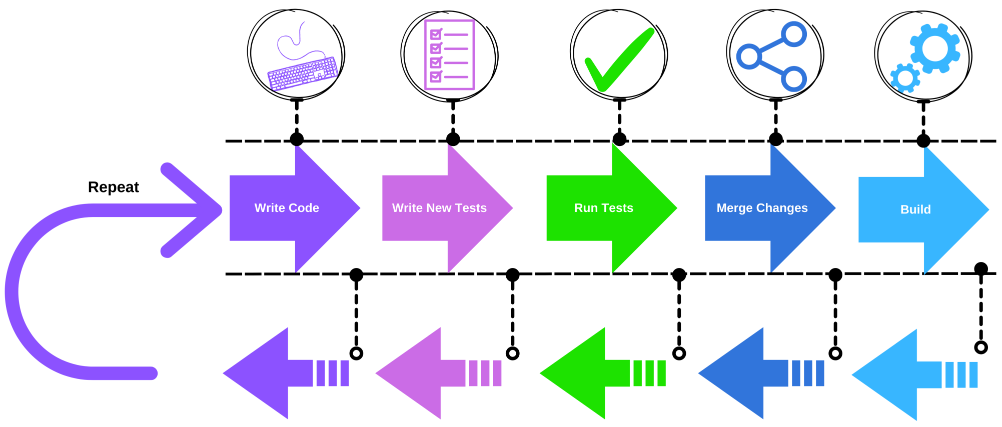


Figure 2.9: Continuous Integration Pipeline

integration. By breaking down the development process into multiple smaller merges before assembling the full code, the AI exhibited a robust capacity to detect errors at various stages of integration. Moreover, due to the granularity of these smaller merges, it was notably easier to identify and correct errors, leading to expedited debugging and refinement of the codebase.

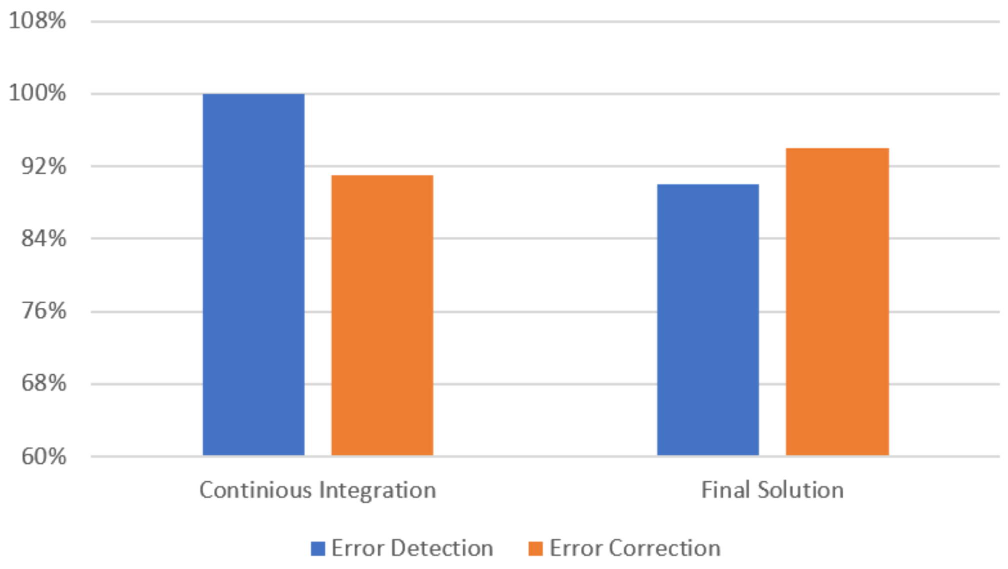


Figure 2.10: Error Detection and Correction in CI

As depicted in Figure 2.10, during CI the error detection rate was significantly higher, with all errors being detected (100%), compared to when providing the final solution, where only 90% of errors were detected. Interestingly, when it comes to error correction, both scenarios showed similar performance. This discrepancy in error detection rates can be attributed to the proactive nature of CI, where errors are identified and addressed at an earlier stage of development, leading to more comprehensive error detection and easier debugging.

### 2.5.3 Developer Productivity and Test Case

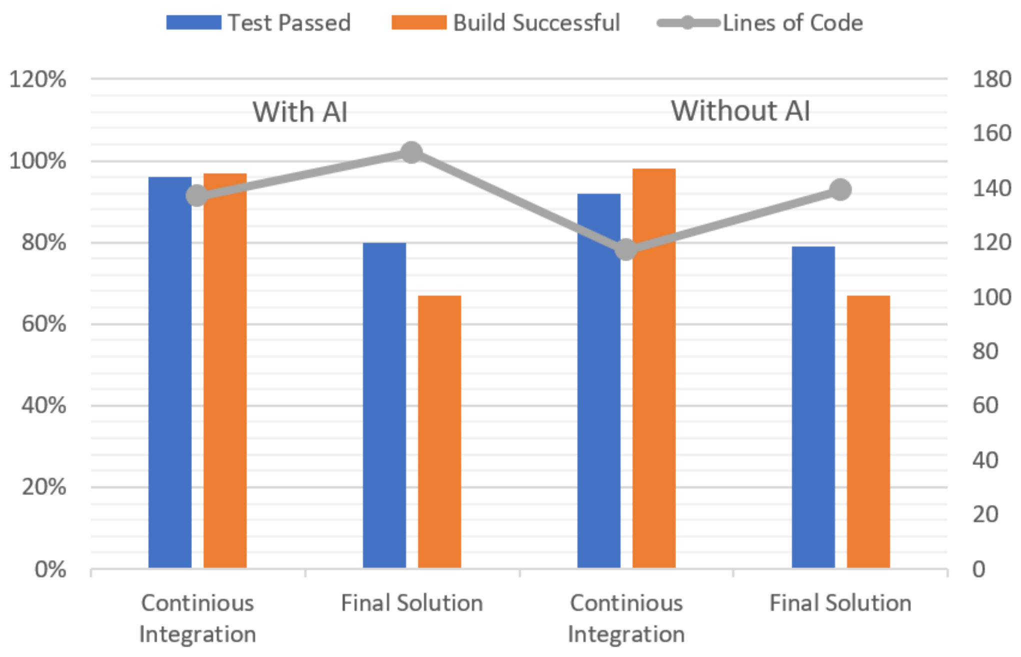


Figure 2.11: CI Evaluation: Human vs AI

In terms of developer productivity and test case outcomes, it's noteworthy that while CI showed a slight regression in productivity compared to the final solution, there was a marked improvement in test and build success rates with CI, irrespective of whether the coding was done by humans or AI. In the Final

Solution scenario, only 2 out of 3 builds were successful, and the test pass rate with AI was at 80%, a similar result was observed with human developers. These findings underscore the advantages of CI as demonstrated in Table 2.11. Furthermore, there's a perceptible improvement with AI assistance.

### 2.5.4 Impact on Software Quality

The analysis of CI implementation reveals a notable impact on software quality. The comparison between CI and the final solution indicates that CI leads to higher rates of test case and build success, also the detection of errors is notably higher compared to the final solution (Figure 2.10), demonstrating its efficacy in ensuring the stability and reliability of the software product. Despite a slightly lower developer productivity observed in CI (Figure 2.11), the improvement in test and build success rates outweighs this drawback, highlighting CI's overall positive impact on software quality. Moreover, the incorporation of AI assistance, particularly in pair programming scenarios, showcases a small enhancement in software quality.

## 2.6 Use Case and User Story

Use cases and user stories are both techniques used in software development to capture requirements and describe how a system will be used from an end-user perspective, but they serve different purposes and have distinct characteristics [15], [44]. Usually when the costumer meets the development team he presents the requirements in the form of user stories or use cases, the team then estimates the stories and make a plan to cover all the requested functionalities.

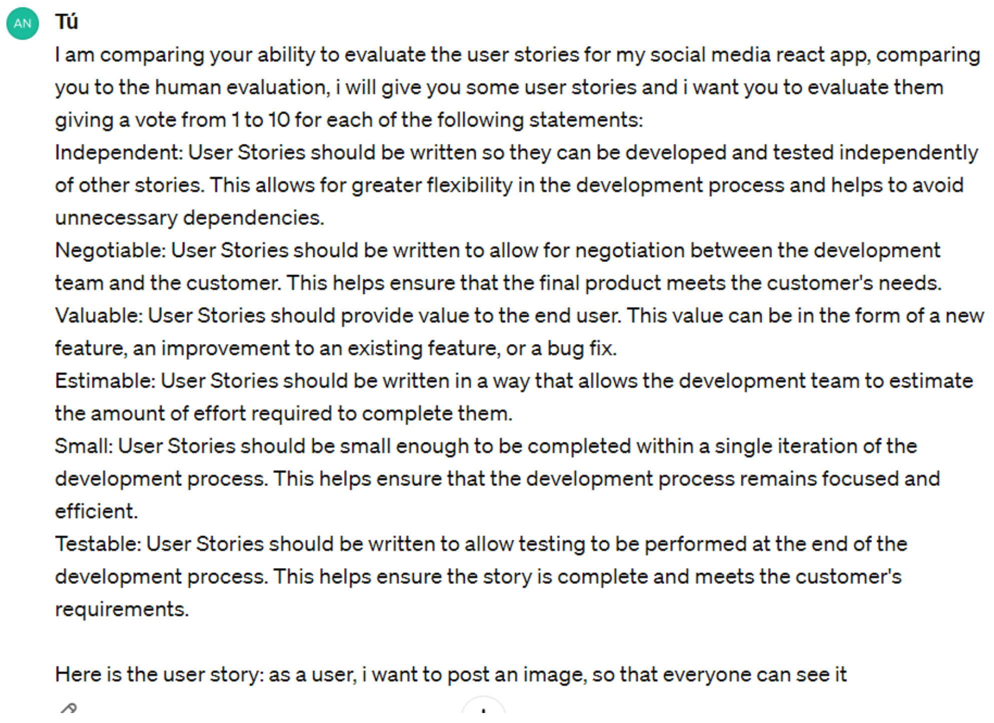
Even thought they are quite similar there is difference in the two practices, user stories are concise, informal descriptions of a feature, it is an XP practice that prioritize conversation over documentation and focus on the "who," "what," and "why" of a feature rather than detailed specifications. They are often written following a simple template:

"As a [**user**], I want [**goal**], so that [**benefit**]."

Use cases, instead, provide a more structured and detailed representation of system behavior. A use case describes a sequence of interactions between an actor (typically a user) and the system to achieve a specific goal. It specifies every step of the implementation, including preconditions, postconditions, and alternative paths, providing a deeper understanding of system functionality. While both serve to understand user requirements and guide development efforts, they diverge in their level of detail and technicality. The reason why we involved the study of use case in this chapter is due to its detailed and structured representation of system behavior, which can offer insights that are particularly valuable for our study's objectives.

### 2.6.1 User Story

As said before, a user story is an informal, general explanation of a feature written by the end-user or customer. In this section we evaluate the capacity of AI in assessing user stories, we conducted a comparative analysis against the evaluations provided by a group of 8 Computer Science students. This approach enabled us to evaluate the AI's effectiveness in estimating user stories compared to human and to understand the efficiency of AI-driven evaluations in terms of time management. Moreover recent studies demonstrated that "The quality of user stories is crucial to the success of a development project as they impact the quality of the system design which, in turn, affects the final product" [36], [4]. Moreover a well-written user story can be submitted to LLMs to generate the corresponding code necessary to fulfill the requirements outlined in the story [17].
The evaluation of each user story was done by following the guidelines of the INVEST criteria [28], which examines how independent, negotiable, valuable, estimable, small and testable the story is. This evaluation criteria was also given to GPT-3.5 in it's prompt, as depicted in Figure 2.12.



Tú

I am comparing your ability to evaluate the user stories for my social media react app, comparing you to the human evaluation, i will give you some user stories and i want you to evaluate them giving a vote from 1 to 10 for each of the following statements:
Independent: User Stories should be written so they can be developed and tested independently of other stories. This allows for greater flexibility in the development process and helps to avoid unnecessary dependencies.
Negotiable: User Stories should be written to allow for negotiation between the development team and the customer. This helps ensure that the final product meets the customer's needs.
Valuable: User Stories should provide value to the end user. This value can be in the form of a new feature, an improvement to an existing feature, or a bug fix.
Estimable: User Stories should be written in a way that allows the development team to estimate the amount of effort required to complete them.
Small: User Stories should be small enough to be completed within a single iteration of the development process. This helps ensure that the development process remains focused and efficient.
Testable: User Stories should be written to allow testing to be performed at the end of the development process. This helps ensure the story is complete and meets the customer's requirements.

Here is the user story: as a user, i want to post an image, so that everyone can see it

Figure 2.12: User Story Prompt

#### 2.6.1.1 User Story Evaluation

From Table 2.3 and Figure 2.13, we can observe several trends and differences between the AI evaluations and human evaluations of the user stories. While both representations convey the same data, Figure 2.13 provides a clearer visualization of the disparities between human and AI evaluations for each user story. The visual depiction enhances the understanding of how human and AI assessments diverge across various aspects of the stories. In general, the AI evaluations tend to rate user stories higher in terms of independence and negotiability compared to human evaluations. This suggests that the AI may be more inclined to view user stories as independent and negotiable, potentially due to its algorithmic nature and lack of subjective biases. Since most of the user stories were correlated, as part of the same react component, the evaluations provided by students regarding the independence criterion were consistently low. Considering that the user stories

were extracted from a university project specification, it's important to note that the concept of negotiability may not have been applicable in this context. As such, the lower scores in negotiability provided by students could be attributed to the constraints of the project requirements rather than a reflection of the user stories' negotiability.
The evaluations for the criteria of valuable, estimable, testable, and small exhibited relatively similar scores across both AI and human evaluations, indicating a consistent perception of these aspects of the user stories even thought there were a few exceptions where discrepancies were observed. The observed similarity in the assessment of these criteria may be attributed to the more technical nature of the evaluation process for these aspects compared, for example, to the negotiability.

Table 2.3: User Story Evaluations

| **Story** | **Evaluation** | **Independent** | **Negotiable** | **Valuable** | **Estimable** | **Small** | **Testable** |
|---|---|---|---|---|---|---|---|
| 1 | AI Evaluation | 9 | 8 | 10 | 9 | 7 | 8 |
| | Human Evaluation | 5 | 2 | 8 | 7 | 6 | 8 |
| 2 | AI Evaluation | 9 | 8 | 10 | 9 | 8 | 9 |
| | Human Evaluation | 2 | 1 | 9 | 7 | 7 | 9 |
| 3 | AI Evaluation | 9 | 8 | 10 | 9 | 8 | 9 |
| | Human Evaluation | 4 | 8 | 6 | 4 | 8 | 8 |
| 4 | AI Evaluation | 8 | 7 | 9 | 8 | 8 | 9 |
| | Human Evaluation | 5 | 2 | 7 | 8 | 4 | 8 |
| 5 | AI Evaluation | 6 | 8 | 5 | 7 | 4 | 6 |
| | Human Evaluation | 3 | 3 | 9 | 9 | 2 | 9 |
| 6 | AI Evaluation | 7 | 9 | 8 | 8 | 6 | 7 |
| | Human Evaluation | 1 | 4 | 8 | 7 | 7 | 7 |
| 7 | AI Evaluation | 8 | 9 | 7 | 8 | 8 | 7 |
| | Human Evaluation | 1 | 1 | 8 | 7 | 6 | 7 |

The average difference between human and AI evaluations was calculated in Figure 2.14 using the following formula:

$$\text{Average Difference} = \frac{\sum_{i=1}^{n} |AI_i - \text{Human}_i|}{n}$$

Where for each parameter:

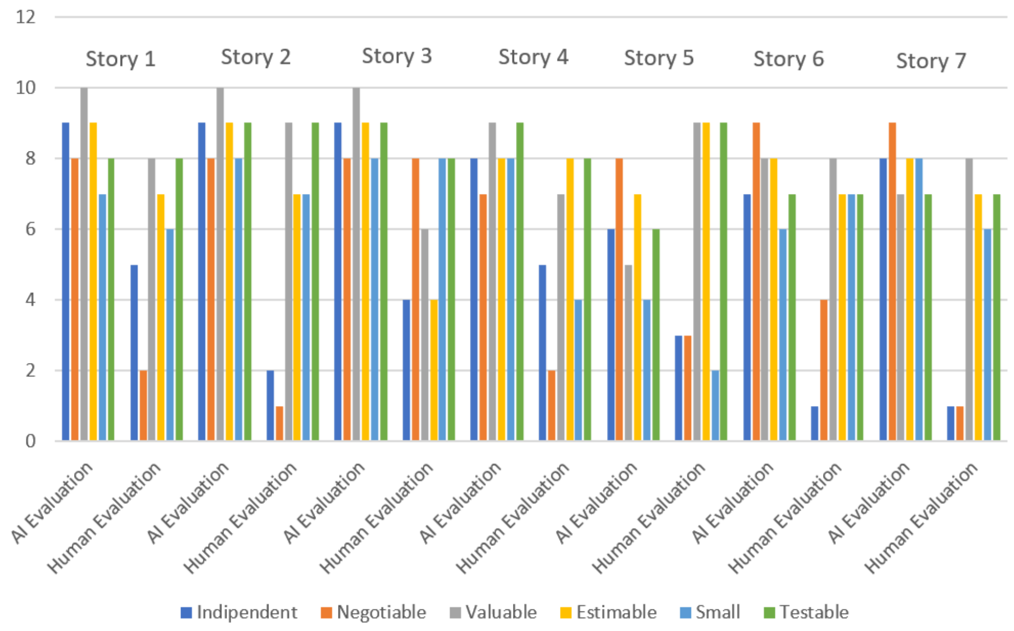


Figure 2.13: User Story Evaluation Plot

- $AI_i$ is the AI evaluation score for user story $i$,
- $Human_i$ is the human evaluation score for user story $i$,
- $n$ is the total number of user stories.

## 2.6.2 Use Case

In this section, we delve into a more technical aspect of software development, use cases. Use cases are a structured way of defining the interactions between users and a system to achieve specific goals. Unlike user stories, which are often expressed in a more narrative form and focus on user needs, use cases are more detailed and provide a step-by-step description of how users interact with the system.
The evaluation of the use cases will be conducted through a structured process. Initially, the AI will be provided with project requirements of a react

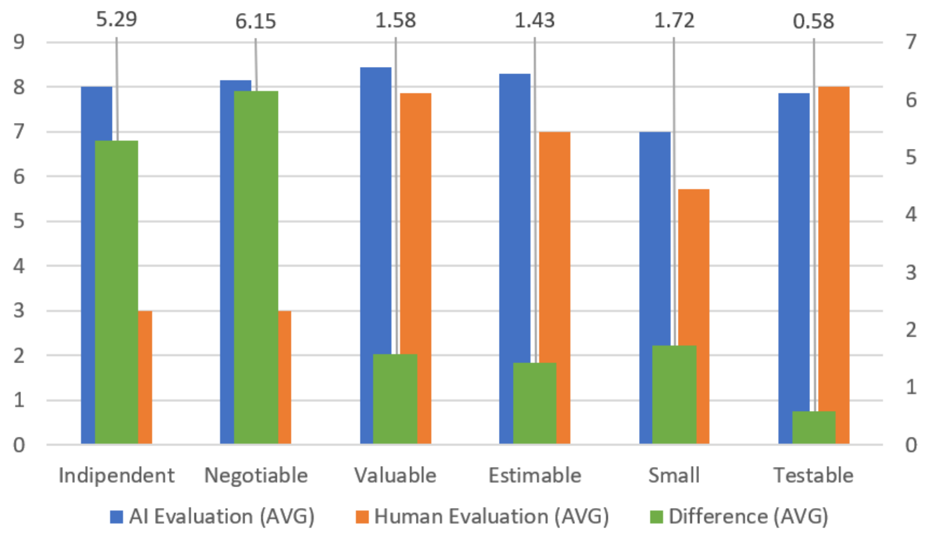
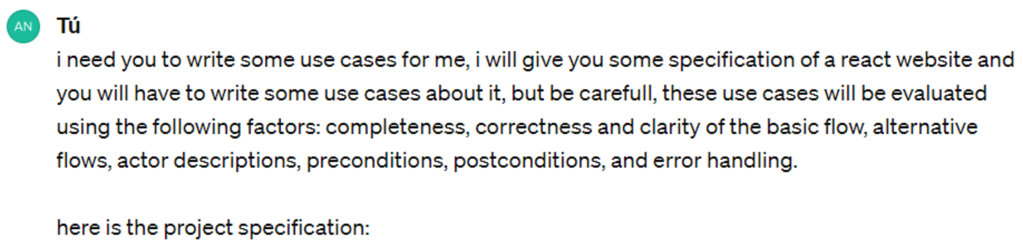


Figure 2.14: User Story Evaluation Plot (AVG)

website and tasked with decomposing them into individual use cases. The prompt used is the following Figure:



**Tú**

i need you to write some use cases for me, i will give you some specification of a react website and you will have to write some use cases about it, but be carefull, these use cases will be evaluated using the following factors: completeness, correctness and clarity of the basic flow, alternative flows, actor descriptions, preconditions, postconditions, and error handling.

here is the project specification:

Figure 2.15: Use Case Prompt

These use cases will then be evaluated according to the criteria outlined in a recent research study [12], which includes factors such as completeness, correctness and clarity of the basic flow, alternative flows, actor descriptions, preconditions, postconditions, and error handling. The results of the evaluation can be observed in Figure 2.16, while Figure 2.17 illustrates the average evaluation score for each criterion, allowing for easy comparison and analysis of each use case's performance against the defined criteria.

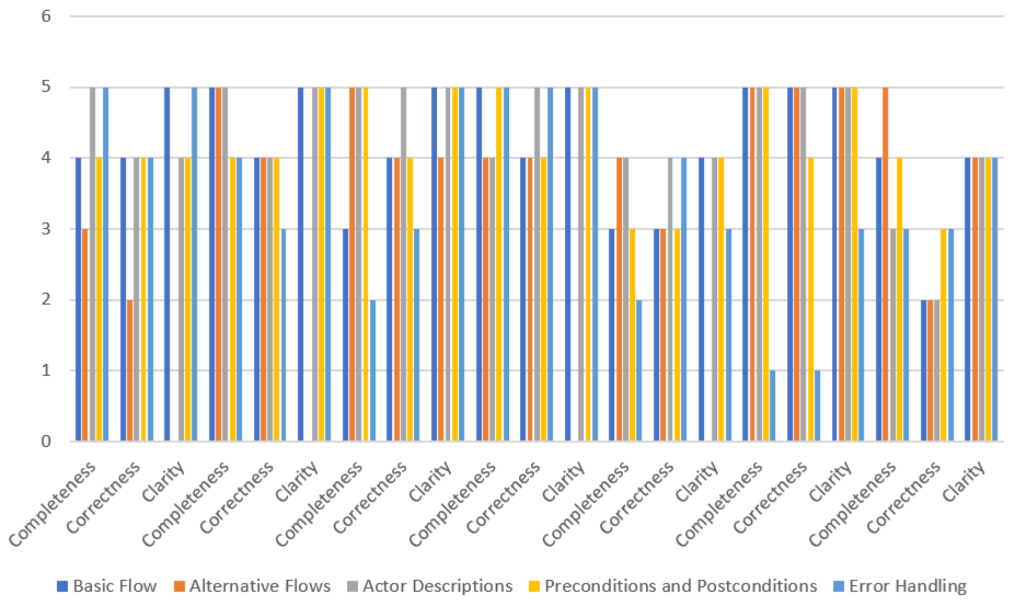


Figure 2.16: Use Cases Evaluations

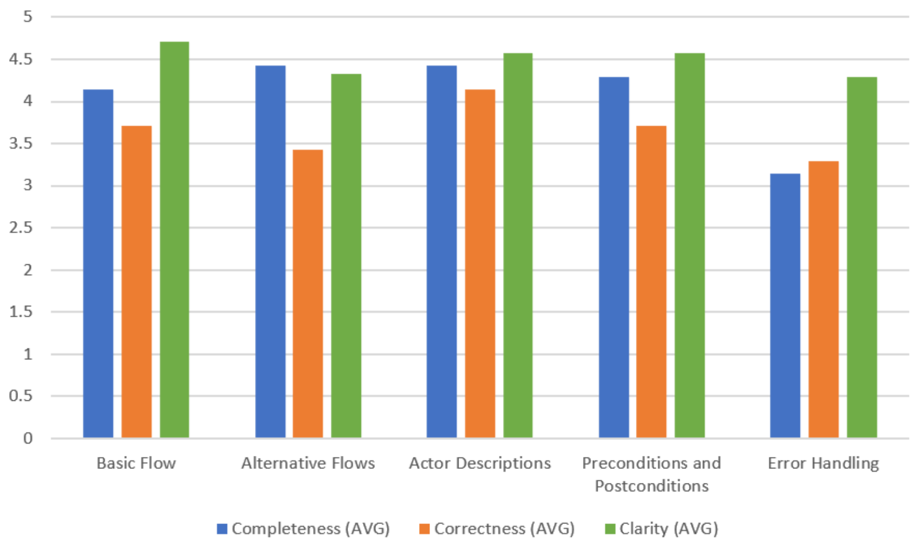


Figure 2.17: Use Cases Evaluations (AVG)

#### 2.6.2.1 Use Case Evaluation

From Figure 2.17, it's evident that AI performed exceptionally well in terms of clarity, with each criterion surpassing a score of 4 (out of 5) in the evaluation. This indicates that the use cases generated by AI were written in a clear and understandable manner, facilitating effective communication of system interactions and requirements. Clarity is a crucial aspect of use case documentation as it ensures that stakeholders can easily comprehend the intended functionality and behavior of the system. The consistently high scores across all clarity-related criteria reflect the effectiveness of AI in producing use cases that are coherent, concise, and comprehensible, thereby contributing to better project understanding and collaboration among stakeholders.
In addition to clarity, AI also demonstrated excellence in actor descriptions, as indicated by the high scores across all related criteria in Figure 2.17. Actor descriptions play a pivotal role in use case documentation as they define the roles and responsibilities of various system users or entities involved in the interaction. The robustness of AI-generated actor descriptions suggests that the system accurately identified and delineated the roles of different actors.
While AI exhibited strength in clarity and actor descriptions, it faced challenges in ensuring the correctness of both the basic and alternative flows. The correctness of these flows is crucial for ensuring that the system behaves as intended and handles various scenarios effectively. The lower scores in correctness for both basic and alternative flows indicate potential discrepancies or inaccuracies in how the AI-generated use cases depict the sequence of actions and decision points. These inaccuracies could lead to misunderstandings or misinterpretations during the development process, potentially resulting in errors or unintended behavior in the implemented system.
Furthermore, the performance of AI in error handling, was also improvable. Error handling is a critical aspect of software systems, as it determines how effectively the system detects, reports, and recovers from unexpected or erroneous situations. The lower scores in error handling suggest that the AI-generated use cases may not adequately address potential errors or excep-

tions that could occur during system operation. Insufficient error handling can lead to situations where the system fails to provide meaningful feedback to users when errors occur, resulting in frustration or confusion.

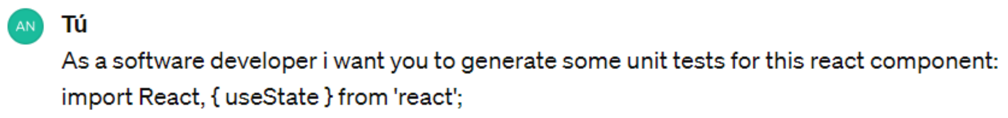
## 2.7 Unit Tests

Unit tests are a fundamental aspect of software development [43] aimed at verifying the correctness of individual units or components of a software [6]. These tests focus on examining the behavior of small, isolated parts of the software, such as functions, methods, or classes. The purpose of unit tests is to ensure that each unit of code performs its intended function correctly under various conditions and inputs [40]. Unit tests are typically automated, meaning they can be executed automatically without manual intervention. They are often written by developers alongside the code they are testing and are executed frequently, typically as part of CI. Given the labor-intensive nature of manually creating unit tests, as highlighted by previous research [37], we will study the use of LLMs to generate these tests.

### 2.7.1 Unit Test Generation

For our study, we employed GPT-3.5 to generate unit tests aimed at evaluating their correctness. It's crucial to emphasize that all the code subjected to these tests was fully functional and thoroughly tested beforehand. The primary objective of this experiment was to assess the AI's capability to produce accurate and beneficial unit tests, rather than to validate the correctness of the existing codebase. This distinction underscores the focus on evaluating the AI's effectiveness in generating tests that contribute to the overall quality and reliability of the software.

Figure 2.18 shows the prompt used to generate unit tests:


Tú

As a software developer i want you to generate some unit tests for this react component:
import React, { useState } from 'react';

Figure 2.18: Unit Test Prompt
(The corresponding code can be found in the appendix)

Table 2.4 will showcase various aspects, including the types and the number of the tests, the number of tests that passed successfully, and the count of comments provided for each file.

Table 2.4: Unit Tests Evaluation Table

| File | Function Tests | Rendering Tests | Other Tests | Total Tests | Passed Tests (%) | No. Of Comments |
|---|---|---|---|---|---|---|
| Form.js | 19 | 11 | 9 | 39 | 24 (61%) | 197 |
| Auth.js | 8 | 8 | 5 | 21 | 20 (95%) | 83 |
| Menu.js | 15 | 4 | 5 | 24 | 19 (79%) | 87 |
| PostDetails.js | 6 | 6 | 3 | 15 | 14 (93%) | 45 |
| SelectSMM.js | 5 | 3 | 3 | 11 | 11 (100%) | 28 |
| Settings.js | 8 | 5 | 3 | 16 | 13 (81%) | 21 |

### 2.7.2 Unit Test Evaluation

Our tests delivered satisfactory performance, with an overall passing rate of 85%. This indicates that a significant majority of our tests (101 out of the 126 generated) executed successfully. Although an 85% pass rate on unit tests may seem satisfactory, it does not fully reflect the AI's capability to consistently generate accurate tests. While the majority of tests may pass, the accuracy and correctness of the generated tests are essential for assessing the AI's proficiency in this task. As illustrated in Table 2.4, the AI demonstrated exemplary performance in providing explanations for each unit test, as evidenced by the comprehensive commenting accompanying each test. In fact, the strength of the AI lies in its meticulous approach to commenting on every individual unit test with remarkable precision, with an average of over three comments per test (Table 2.5), it demonstrates a commitment to thorough and detailed documentation, enhancing the clarity and understanding of the tests.

Table 2.5: Summary of Test Results

| Total Tests | Passed Tests | Passed Tests (%) | AVG Comment Count per Test |
|---|---|---|---|
| 126 | 101 | 85.09% | 3.247 |

# Chapter 3

# Discussion

In this chapter, we delve into a comprehensive analysis of the findings obtained from our study, addressing the research questions posed earlier. We critically evaluate the performance of LLMs in the context of agile software engineering practices, highlighting their impact on productivity, code quality, and collaboration among developers. Additionally, we explore the challenges and limitations associated with the adoption of LLMs in agile environments, shedding light on potential areas for improvement and future research directions. Through an examination of both the strengths and weaknesses of LLM integration, we aim to provide valuable insights into the effective utilization of these powerful AI tools in agile software development.

## 3.1 Analysis of Findings

In analyzing the findings of this study, it becomes evident that the integration of LLMs into agile software engineering practices has yielded a predominantly positive impact. Additionally, other studies have highlighted the potential for integrating LLMs (ChatGPT) into the software engineering workflow [1]. These investigations sustain our findings and underscore the broader applicability of LLMs in enhancing various aspects of software development processes. One of the most notable advantages observed in our study is the significant enhancement of productivity, particularly evident in Test-Driven Development and pair programming scenarios. Moreover, LLMs have played a key role in automating tasks sometimes perceived as repetitive and time-consuming, such as code refactoring and testing.

Through our investigation, we discovered a key strategy for harnessing the capabilities of LLMs effectively: dividing requests into smaller, more manageable tasks rather than inundating the model with numerous operations simultaneously. This approach was decisive in optimizing the performance and response accuracy of LLMs within the context of agile software engineering practices.

By breaking down requests into smaller tasks, developers can mitigate the risk of overwhelming the model and ensure that it can devote adequate attention and resources to each operation. Moreover, dividing requests into smaller tasks aligns with agile development principles, facilitating a more iterative and incremental approach to problem-solving. It allows developers to address specific aspects of a problem or task incrementally, iteratively refining and adjusting their approach based on feedback and intermediate results.

While our analysis revealed significant improvements across various aspects of XP, particularly in areas such as test generation, code refactoring, and the quality of generated code, our analysis also revealed areas for improvement and potential challenges that need to be addressed.

In the domain of unit testing, we observed that while LLMs were proficient in generating test cases, there was a notable discrepancy in the effectiveness of these tests. Specifically, our findings indicated that only 85% of the generated tests were functioning correctly, highlighting the need for further refinement and validation of test cases to ensure comprehensive coverage and accuracy.

Similarly, our analysis of code quality revealed that code generated by LLMs exhibited lower quality compared to human-written code, particularly when evaluated against a predefined set of criteria. Additionally, it's important to highlight that the generated code frequently requires human validation to

guarantee it's functionality, a point highlighted in previous studies such as [33].
Furthermore, in the domain of code refactoring, we encountered instances where automated refactoring performed by LLMs resulted in unintended consequences, such as changes to functionality or even code breakages. This phenomenon contradicts the fundamental principles of refactoring, which aim to improve code structure and design without altering external behavior [16].

## 3.2 Research Answers

### 3.2.1 RQ1: How can LLMs be effectively integrated into agile software development practices to improve productivity, code quality, and collaboration among developers?

To answer RQ1, our study offers several key insights and recommendations based on our findings.

#### Task Segmentation

Our research underscores the importance of breaking down development tasks into smaller, manageable units to leverage the capabilities of LLMs effectively. By dividing tasks and focusing on specific operations, developers can optimize the utilization of LLMs and minimize the risk of errors or inefficiencies.

#### Automation of Repetitive Tasks

LLMs can be very effective in automating repetitive and time-consuming tasks, such as test case generation and documentation. Integrating LLMs into agile workflows can streamline these processes, freeing up developers' time to focus on higher-level design and problem-solving activities.

#### Collaborative Development Environment

By leveraging LLMs as virtual teammates, developers can enhance their collaborative problem-solving sessions. Here, AI assistants contribute insights, suggest solutions, and provide real-time feedback on code implementations, thus significantly boosting productivity.

### 3.2.2 RQ2: What are the potential challenges and limitations associated with the adoption of LLMs in agile software engineering contexts, and how can they be mitigated?

To answer RQ2, we must acknowledge several potential challenges and limitations associated with the adoption of LLMs in agile software engineering contexts:

**Quality Assurance**

One prominent challenge is ensuring the quality and reliability of code generated by LLMs. As highlighted in our findings, there is a notable discrepancy between human-generated code and AI-generated code in terms of quality. Mitigating this challenge requires implementing robust quality assurance processes or even refining LLM training methodologies to produce higher-quality outputs.

**Functional Changes during Refactoring**

Another significant limitation observed is the risk of unintended functional changes or code breakages during the refactoring process. This poses a fundamental challenge to the definition of refactoring and undermines its purpose.

**Test Reliability**

Our research uncovered that not all of the generated unit tests produced by LLMs were functioning correctly. This highlights a crucial challenge in ensuring the reliability and effectiveness of test suites generated by AI. To mitigate this limitation, comprehensive testing protocols and continuous validation processes should be established to verify the accuracy and coverage of generated tests.

**Adaptation to Specific Contexts**
LLMs may struggle to adapt effectively to the specific contexts and requirements of individual software projects. This lack of adaptability can impede their usefulness and effectiveness in certain scenarios.

## 3.3 Open Problems, Limitations and Future Work

It is essential to acknowledge that while the benefits of LLM integration inside an agile development team are considerable, challenges and limitations exist. In this context, two significant challenges warrant attention: the security implications of AI-generated code and the phenomenon of hallucination in language models. While these issues have not been thoroughly explored in this study, they represent important areas for further investigation and refinement. The security of AI-generated code raises concerns regarding vulnerabilities and potential exploits that may arise from automated code generation processes. Recent studies, such as [19], [24], [26] and [35], have highlighted various vulnerabilities and security risks associated with code generated by LLMs. These findings underscore the importance of addressing security considerations in the adoption of LLMs in agile software development practices.
Additionally, the problem of hallucination pertains to instances where language models produce erroneous or misleading outputs, which could pose risks in critical software development contexts, as highlighted in recent studies [48], [13].
Furthermore, the issues of data privacy, misinformation, and ownership now come to the forefront. The reliance on vast amounts of data raises concerns

about user privacy and the potential for data misuse and the proliferation of AI-generated content raises the risk of misinformation dissemination as studied in [21]. Moreover, questions surrounding intellectual property rights and ownership arise when AI systems generate code or content based on proprietary data or algorithms [32]. Addressing these challenges is essential for ensuring ethical and responsible use of AI technologies in software engineering.

In addition, it is important to acknowledge the limitation of the relatively small number of human developers we have involved in our experiments. A wider range of participants with diverse backgrounds and expertise levels could potentially yield varied results, offering a more comprehensive understanding of the implications of LLM integration across different contexts. Future research could explore this aspect by involving a more extensive and diverse pool of human developers, allowing for a more significant analysis of the effectiveness and challenges associated with LLM integration in agile software development.

In conclusion, an additional area for future research lies in refining the prompts provided to LLMs to enhance their performance in agile software development. Recent studies, such as [46], suggest that optimizing prompt quality could lead to significant improvements in LLMs' effectiveness in this domain. Thus, exploring strategies to enhance prompt design and utilization represents a promising avenue for advancing the integration of LLMs into agile practices.

# Chapter 4

# Conclusion

In this thesis, our primary objective was to explore the integration of Large Language Models (LLMs) into agile software engineering practices and evaluate their potential impact on eXtreme Programming (XP). Now, having examined the results of our research, we can delve into the key findings and implications of our study.
Our analysis of the integration of LLMs into agile software engineering practices underscores the significant potential for advancements in productivity and code quality, aligning with the findings presented in a recent thesis work [18]. Through a mixed-methods approach combining quantitative and qualitative analysis, we have gained valuable insights into the benefits and challenges associated with LLM integration in this context. Our study highlights promising advancements, particularly in Test-Driven Development and collaborative programming scenarios, where LLMs demonstrate efficacy in automating repetitive tasks and enhancing efficiency. However, we recognize the importance of addressing challenges such as the need for human validation of LLM-generated code to ensure accuracy and reliability, as evidenced by instances of code quality discrepancies compared to human-written code. Despite these challenges, the overall impact of LLMs on agile software development appears positive, with the potential to drive significant improvements in efficiency, collaboration, and code quality. Moving forward, it is imper-

ative for researchers and practitioners to continue refining LLMs prompts, addressing security concerns, and exploring innovative ways to harness the full potential of LLMs in agile development environments. Ultimately, our research contributes to advancing the understanding of how LLMs can augment human developers' capabilities within agile practices, offering valuable insights for future research and practical implementation in software development contexts.

# Appendix

## Unit Test Generation Code

This code snippet serves as an illustrative example provided to the AI for the purpose of unit test generation.

```
import React, { useState, useEffect } from 'react';
import { Container, Grow, Grid, Paper, AppBar, TextField, Button,
Typography } from '@material-ui/core';
import { useDispatch } from 'react-redux';
import Select from 'react-select';
import { getSMMs, setSMM, getMySMM } from '../../actions/auth';
import { ToastContainer, toast } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';

import useStyles from '../styles';

function SelectSmm() {
    const classes = useStyles();
    const user = JSON.parse(localStorage.getItem('profile'));
    const dispatch = useDispatch();
    const [smms, setSmms] = useState([]);
    const [smm, setSmm] = useState('');

    const handleSelectUsers = (selectedOption, actionMeta) => {
        setSmm(selectedOption);
    }

    const clearSMM = () => {
        setSmm('');
    }



    const handleSubmitSMM = async (e) => {
        e.preventDefault();

        dispatch(setSMM(user.result._id, (smm.value ? smm.value
        : '')));
        toast("Done!", { type: "success" });
        getSMM();
    }

    const getSMM = async () => {
        await dispatch(getSMMs()).then((res) => {
            setSmms(res);
        });
    }

    const getMySmm = async () => {
        await dispatch(getMySMM(user.result._id)).then((res) => {
            setSmm(res);
        });
    }

    useEffect(() => {
        if (user?.result?.role !== 'vip') window.location.href =
    window.location.origin + '/react';

    if (user) {
        getMySmm();
        getSMM();
        //setSmms(smms.concat(smm));
        //console.log(smm);
    }
    //console.log(user?.result);
}, []);

return (
    <Container maxWidth="sm">
        <Paper className={classes.paper} elevation={6}>
            <form autoComplete="off" noValidate className=
            {`${classes.root} ${classes.form}`} onSubmit=
            {handleSubmitSMM}>
                <Typography variant="h6">Select SMM
                </Typography>
                <Select className={classes.fileInput}
                options={smms} value={smm} fullWidth onChange=
                {handleSelectUsers} />
                <Button className={classes.buttonSubmit}
                variant="contained" color="primary"
                size="large" type="submit"
                fullWidth>Confirm</Button>
                <Button variant="contained" color="secondary"
                size="small" onClick={clearSMM} fullWidth
                >Remove SMM</Button>
                <ToastContainer autoClose={1000}
                hideProgressBar={true} />
                </form>
            </Paper>
        </Container>
    );
}

export default SelectSmm;
```

## Refactoring Code

This code snippet serves as an illustrative example provided to the AI for the purpose of code refactoring.

```
import React, { useState, useRef } from 'react';
import { Avatar, Button as Butt, Paper, Grid, Typography, Container }
from '@material-ui/core';
import LockOutlinedIcon from '@material-ui/icons/LockOutlined';
import { useDispatch } from 'react-redux';
import { useNavigate } from 'react-router-dom';
import useStyles from './styles';
import Input from '../Auth/Input';
//import Icon from './icon';
import { ConfirmDialog, confirmDialog } from
'primereact/confirmdialog';
import { Button } from 'primereact/button';
import { Toast } from 'primereact/toast';
import { ToastContainer, toast } from 'react-toastify';
import { updatePassword, deleteAccount } from '../../actions/auth';

import 'primeicons/primeicons.css';
//theme
import "primereact/resources/themes/lara-light-indigo/theme.css";
```

```
//core
import "primereact/resources/primereact.min.css";



const initialState = { oldPassword: '', newPassword: '',
confirmPassword: '' };

const Settings = () => {
    const [user, setUser] = useState(JSON.parse(
    localStorage.getItem('profile')));
    const classes = useStyles();
    const [showPassword1, setShowPassword1] = useState(false);
    const [showPassword2, setShowPassword2] = useState(false);
    const [formData, setFormData] = useState(initialState);
    const [deletingAccount, setDeletingAccount] = useState(false);
    const toast = useRef(null);
    const dispatch = useDispatch();
    const navigate = useNavigate();

    const handleShowPassword1 = () => setShowPassword1(
    (prevShowPassword1) => !prevShowPassword1);
    const handleShowPassword2 = () => setShowPassword2(
    (prevShowPassword2) => !prevShowPassword2);

    const handleSubmit = (e) => {
        e.preventDefault();

        if (formData.newPassword === formData.confirmPassword) {
            dispatch(updatePassword(user?.result?._id, formData,
            navigate));
            toast.current.show({ severity: 'success', summary:
```

```
        'Confirmed', detail: 'Password Changed!', life:
        7000 });
        //alert("Password Changed!", { type: "success" });
    } else {
        alert("Passwords don't match");
    }

};

const logout = () => {
    dispatch({ type: 'LOGOUT' });
    setUser(null);
    navigate('/');
}

const accept = () => {
    toast.current.show({ severity: 'info', summary: 'Confirmed',
    detail: 'Account Deleted!', life: 3000 });
    setDeletingAccount(true);
    dispatch(deleteAccount({ _id: user?.result?._id }, navigate));

    setTimeout(() => {
        //dispatch(deleteAccount({ _id: user?.result?._id },
        navigate));
        logout();
    }, 3000);
    //logout();
}

const reject = () => {
    toast.current.show({ severity: 'warn', summary: 'Rejected',
    detail: 'Operation Canceled', life: 3000 });
}

const confirm2 = () => {
    confirmDialog({
        message: 'This operation is irreversible. Do you want
        to proceed?',
        header: 'Delete Confirmation',
        icon: 'pi pi-exclamation-triangle',
        acceptClassName: 'p-button-danger',
        accept,
        reject
    });
};

const handleChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
};

return (
    <Container component="main" maxWidth="xs">
        <Paper className={classes.paper} elevation={3}>
            <Toast ref={toast} />
            {deletingAccount ? (
                <>
                    <Typography style={{ marginBottom: '40px',
                    color: 'red' }} variant="h4">Deleting account...
                    </Typography>
                    <i className="pi pi-spin pi-cog" style={{
                    fontSize: '20rem' }}></i>
                </>
```

```
) : (
<>
<Avatar className={classes.avatar}>
    <LockOutlinedIcon />
</Avatar>
<Typography variant="h5">Password Change</Typography>
<form className={classes.form} onSubmit={handleSubmit}>
    <Grid container spacing={2}>
        <Input name="oldPassword" label="Old Password"
        handleChange={handleChange} type={showPassword1 ?
        "text" : "password"}
        handleShowPassword={handleShowPassword1} />
        <Input name="newPassword" label="New Password"
        handleChange={handleChange} type={showPassword2 ?
        "text" : "password"}
        handleShowPassword={handleShowPassword2} />
        <Input name="confirmPassword" label="Confirm
        Password" handleChange={handleChange}
        type={showPassword2 ? "text" : "password"} />

    </Grid>
    <Butt type="submit" fullWidth variant="contained"
    color="primary" className={classes.submit} >
        Change Password
    </Butt>
    <ToastContainer autoClose={1000}
    hideProgressBar={true} />
</form>

<ConfirmDialog />
<div className="card flex flex-wrap gap-2
```

```
                justify-content-center">
                    <Button onClick={confirm2} icon="pi pi-times"
                    label="Delete Account" severity="danger"
                    outlined></Button>
                </div>
                </>
                )}

            </Paper >

        </Container >
    );
}

export default Settings;
```

# Ringraziamenti

Desidero esprimere il mio sincero ringraziamento al mio relatore per la sua preziosa guida e supporto durante la redazione di questa tesi.

Ai miei genitori, che hanno sempre creduto in me e hanno sacrificato tanto per farmi avere le migliori opportunitá possibili, non potró mai ringraziarvi abbastanza. La vostra dedizione e il vostro incoraggiamento mi hanno dato la forza di perseguire i miei sogni e di raggiungere questo importante traguardo.

Ringrazio di cuore i miei nonni per avermi deliziato con i loro piatti durante il mio periodo all'universitá. La vostra cucina é stata una fonte di conforto e gioia che ha reso i giorni di studio molto piú piacevoli. La vostra premura e il vostro amore sono stati veramente preziosi per me.

Voglio dedicare un ringraziamento speciale alla mia fidanzata per essere stata al mio fianco durante questo emozionante periodo. Le nostre avventure, risate e il tempo trascorso insieme hanno reso ogni sfida piú leggera. Senza di te, non avrei potuto raggiungere questo traguardo.

Un ringraziamento particolare va a coloro che hanno contribuito alla raccolta e all'analisi dei dati presenti in questa tesi, collaborando con me nello svolgimento degli esercizi di programmazione.

Non posso dimenticare di ringraziare tutte le persone straordinarie che ho avuto il privilegio di incontrare durante il mio periodo di scambio in Turchia. È stata un'esperienza indimenticabile che ha arricchito profondamente il mio percorso accademico e personale.

Desidero esprimere la mia sincera gratitudine ai genitori della mia fidanzata per avermi aperto le porte della loro casa durante il periodo di scrittura della mia tesi. La loro generosa ospitalitÃ  ha reso possibile per me condividere questo importante momento accanto alla mia amata

Infine, desidero esprimere la mia gratitudine a tutte le persone fantastiche che ho avuto l'onore di conoscere lungo questo affascinante percorso. Il vostro contributo e il vostro sostegno hanno reso possibile il completamento di questa tesi. Grazie di cuore.

# Bibliography

[1] Pekka Abrahamsson, Tatu Anttila, Jyri Hakala, Juulia Ketola, Anna Knappe, Daniel Lahtinen, Väinö Liukko, Timo Poranen, Topi-Matti Ritala, and Manu Setälä. ChatGPT as a Fullstack Web Developer - Early Results. In *Agile Processes in Software Engineering and Extreme Programming Workshops*, volume 489 of *LNBIP*, pages 201–209. Springer, 2023. URL: https://link.springer.com/chapter/10.1007/978-3-031-48550-3_20.

[2] Nadeem Ahmad. The Power of Pair Programming: Boosting Code Quality and Collaboration, 2023. Medium, Accessed January 18, 2023. URL: https://medium.com/@nadeem.ahmad.na/the-power-of-pair-programming-boosting-code-quality-and-collaboration-6e473b6a5a7d.

[3] Amazon. What is CI? - Continuous Integration Explained. Accessed February 12, 2024. URL: https://aws.amazon.com/devops/continuous-integration/.

[4] A. R. Amna and G. Poels. Systematic Literature Mapping of User Story Research. *IEEE Access*, 10:51723–51746, 2022. doi:10.1109/ACCESS.2022.3173745.

[5] The Content Authority. Devops words - 101+ words related to devops, 2024. Accessed February 15, 2024. URL: https://thecontentauthority.com/blog/words-related-to-devops.

[6] N. Bakharev. Unit Testing: Definition, Examples, and Critical Best Practices, 2023. URL: https://brightsec.com/blog/unit-testing/.

[7] A. Bartolo. Artificial intelligence pair programming with github copilot, 2024. LinkedIn, Accessed January 15, 2024. URL: https://www.linkedin.com/pulse/artificial-intelligence-pair-programming-github-copilot-bartolo/.

[8] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2000.

[9] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.

[10] R. Bellairs. What is code quality? overview, 2019. Accessed February 9, 2024. URL: https://www.perforce.com/blog/sca/what-code-quality-overview.

[11] E. Chen, R. Huang, J. Liang, D. Chen, and P. Hung. GPTutor: An open-source AI pair programming tool alternative to Copilot, 2023. arXiv:arXiv:2310.13896.

[12] G. De Vito, F. Palomba, C. Gravino, S. Di Martino, and F. Ferrucci. ECHO: An approach to enhance use case quality exploiting large language models. In *Proc. 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 53–60, 2023. doi:10.1109/SEAA60479.2023.00017.

[13] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems, 2023. arXiv:arXiv:2310.03533.

[14] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017. doi:10.1016/j.jss.2015.06.063.

[15] M. Fowler. Use cases and stories, 2003. URL: https://martinfowler.com/bliki/UseCasesAndStories.html.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[17] Oscar Garcia. AI Engineering Generate Code from User Stories, 2023. Accessed December 18, 2023. URL: https://www.ozkary.com/2023/05/ai-engineering-generate-code-from-user-stories.html.

[18] Adam Hörnemalm. ChatGPT as a software development tool: The future of development. Master thesis, Dept. of Applied Physics and Electronics, Umeå University, June 2023. Master of Science in Interaction Technology.

[19] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review, 2023. arXiv:arXiv:2308.10620.

[20] S. Imai. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings)*, pages 319–321, Pittsburgh, PA, USA, 2022. doi:10.1145/3510454.3522684.

[21] S. Jalil. The Transformative Influence of Large Language Models on Software Development, 2023. arXiv:arXiv:2311.16429.

[22] Ron Jeffries. What is extreme programming?, 2011. Accessed February 20, 2024. URL: https://ronjeffries.com/xprog/what-is-extreme-programming/.

[23] A. Kumar. Comprehensive guide to code quality!, 2024. Accessed February 9, 2024. URL: https://www.linkedin.com/pulse/comprehensive-guide-code-quality-arvind-kumar-9bjgc/.

[24] Mohammed Latif Siddiq and Joanna C. S. Santos. Generate and Pray: Using SALLMS to Evaluate the Security of LLM Generated Code, 2023. arXiv:arXiv:2311.00889.

[25] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015. doi:10.1109/MS.2015.50.

[26] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against llm-integrated applications, 2023. arXiv:arXiv:2306.05499.

[27] Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. Is ai the better programming partner? human-human pair programming vs. human-ai pair programming, 2023. arXiv:arXiv:2306.05153.

[28] D. Machado. How to evaluate your user stories using the INVEST criteria, 2023. URL: https://www.towerhousestudio.com/blog/how-to-evaluate-your-user-stories-using-the-invest-criteria.

[29] T. Miyashita, T. Katayama, Y. Kita, H. Yamaba, K. Aburada, and N. Okazaki. Prototype of the framework catdd to support continuous development in test driven development. *Journal of Advances in Artificial Life Robotics*, 4(1):35–40, 2023.

[30] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. doi:10.1109/TSE.2011.41.

[31] H. A. Nekrasov. Coding standards and quality checks: Key factors for successful software development. In *2023 International Conference on Quality Management, Transport and Information Security, Information Technologies (IT&QM&IS)*, pages 19–22, 2023. doi:10.1109/ITQMTIS58985.2023.10346356.

[32] I. Ozkaya, A. Carleton, J. Robert, and D. Schmidt. Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?, 2023. Carnegie Mellon University, Software Engineering Institute's Insights (blog), Accessed December 18, 2023. URL: `https://doi.org/10.58012/6n1p-pw64`.

[33] R. A. Poldrack, T. Lu, and G. Beguš. AI-assisted coding: Experiments with GPT-4, 2023. `arXiv:arXiv:2304.13187`.

[34] Y. Rafique and V. B. Mišić. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2013. `doi:10.1109/TSE.2012.28`.

[35] Abhinav Rao, Sachin Vashistha, Atharva Naik, Somak Aditya, and Monojit Choudhury. Tricking llms into disobedience: Understanding, analyzing, and preventing jailbreaks, 2023. `arXiv:arXiv:2305.14965`.

[36] K. Ronanki, B. Cabrero-Daniel, and C. Berger. ChatGPT as a tool for User Story Quality Evaluation: Trustworthy Out of the Box?, 2023. `arXiv:arXiv:2306.12132`.

[37] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023. `doi:10.1109/TSE.2023.3334955`.

[38] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. `doi:10.1109/ACCESS.2017.2685629`.

[39] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe. Refactoring programs using large language models with few-shot examples, 2023. `arXiv:arXiv:2311.11690`.

[40] O. Stoliarchuk. Managing quality, 2023. URL: https://www.linkedin.com/pulse/managing-quality-oleksandr-stoliarchuk/.

[41] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proc. Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pages 10–10, Minneapolis, MN, USA, 2007. doi:10.1109/WOSQ.2007.11.

[42] A. Tornhill, M. Borg, and E. Mones. Refactoring vs refuctoring: advancing the state of ai automated code improvements, 2024. Accessed February 7, 2024. URL: https://codescene.com/hubfs/whitepapers/Refactoring-vs-Refuctoring-Advancing-the-state-of-AI-automated-code-improvements.pdf.

[43] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2020. arXiv:arXiv:2009.05617.

[44] D. Varga. User story vs use case: Everything you need to know, 2020. URL: https://www.digitalnatives.hu/blog/user-story-vs-use-case/.

[45] J. Wang and Y. Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289, Beijing, China, 2023. doi:10.1109/MedAI59581.2023.00044.

[46] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, 2023. arXiv:arXiv:2303.07839.

[47] Wikipedia. Pair programming, 2023. Accessed December 22, 2023. URL: https://en.wikipedia.org/wiki/Pair_programming.

[48] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, Xia Hu, et al. Harnessing the Power

of LLMs in Practice: A Survey on ChatGPT and Beyond, 2023. arXiv:arXiv:2304.13712.