Design and Evaluation of an AI-Driven Workflow for Technical Debt Remediation in Software Systems

University of Turku
Department of Computing
Master of Science Thesis
Software Engineering
July 2025
Rehan Khalil

UNIVERSITY OF TURKU
Department of Computing

Rehan Khalil: Design and Evaluation of an AI-Driven Workflow for Technical Debt Remediation in Software Systems

Master of Science Thesis, 72 p., 7 app. p.
Software Engineering
July 2025

Technical debt (TD) is a persistent challenge in software engineering. It impacts long-term maintenance, reliability and security of the software as well as developer productivity. Many static analysis tools can identify debt, but taking actions, especially on the large scale, often requires a lot of work. This thesis explores how Large Language Model (LLM)-powered AI systems can automatically fix technical debt in real-world software.

This study investigates the application of AI-assisted tools in improving the maintainability of modern web applications. Using a case study of a progressive web app with identified technical debt, the project involved automated analysis using static code analyzer and security code scanner to detect code quality issues such as code smells, bugs and security vulnerabilities. The collected insights were then used in LLM-powered agentic AI to assess the effectiveness in remediating those issues.

Empirical results show that AI resolved approximately 90% issues, significantly improving the maintainability index of the project, by reducing code smells and bugs, and improving security vulnerabilities. However, the AI faced challenges in solving complexities requiring architectural decision and human validation. The study also show that it is possible to integrate diagnostic tools and AI agents into a continuous maintenance process.

This thesis offers a methodology for automated repayment of technical debt. It provides a critical assessment of AI tools capabilities in software maintenance and a better understanding of sustainable development practices with smart tooling. It concludes that while full autonomy is not yet possible, but LLM-powered tools are a promising step toward more effective and efficient technical debt management and repayment.

Keywords: technical debt, software maintainability, agentic AI, LLMs, SonarQube, Snyk, Cursor, Claude Sonnet, automation, software quality, AI-powered remediation

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

As software systems evolve over time, they often gather hidden cost in the form of technical debt. This term describes the long-term effects of poor or hurried development decisions which arise from low code quality, delayed testing, poor documentation or outdated dependencies. These practices make the software harder to maintain, less reliable, and more expensive to update. Though these kind of decisions may meet short-term business goals, they usually harm software maintenance, create security risks, and increase the cost and complexity even for small future updates.

It becomes essential to address technical debt since it consumes a significant portion of developer's time [1] and costs trillions of dollars annually [2]. Although previous researches has introduced rich classification and types of technical debt, there is still a significant gap in practical solutions to remediate technical debt. In recent years, AI-powered tools have shown promising results not only in detecting technical issues but also possibly in fixing them [3], [4], [5].

This thesis aims to explore whether these emerging AI tools can effectively help with technical debt remediation, especially when combined with static code analysis and security scanning tools commonly used in software engineering.

1.2 Problem Statement

Software development practices lack effective and automated ways to address technical debt once it is identified, creating a significant gap in current tooling and workflows [6]. Even though tools like SonarQube and Snyk are widely used to find code smells, bugs and vulnerabilities, the remediation process is still manual. This process is time consuming and depends on the availability and skill of developers.

The fast paced advancements of tools, libraries and frameworks also create additional challenges particularly in the long-term sustainability and maintainability of the software systems. The issue is particularly evident in the real-world case study examined in this thesis, where an existing system had to be redeveloped due to outdated and unmaintained dependencies, and obsolete architecture which are clear signs of technical debt.

While most of the existing research work focuses on exploring the causes and impact of the technical debt, there is a lack of practical guidance on leveraging modern tools to generate actionable insights and address maintainability challenges effectively.

1.3 Research Objective

The objective of this research is to contribute to the field of sustainable software development specifically around technical debt, and usage of AI tools in the software development life cycle process, by analyzing real-world case study. The goal is to understand and examine how technical debt is introduced, how it can be assessed using modern tools, and how AI-driven approaches, specifically LLM-powered agentic tools, can be utilized to remediate these issues practically.

Though significant research work has been done in classifying and identifying technical debt, there is still a gap to explore automated and AI-driven remediation of

technical debt. This thesis tries to fill that gap by conducting a structured evaluation of AI-driven technical debt repayment on a real-world project with industry-used tools.

1.4 Research Questions

The goal of this thesis is to answer following research questions:

RQ1. What development practices contribute to the accumulation of technical debt in software systems?

RQ2. What strategies can be employed to help reduce long-term maintainability issues in real-world systems?

RQ3. How available modern tools can be utilized for technical debt assessment to provide analysis about maintainability, reliability and security of a software?

RQ4. How effective is an LLM-powered agentic AI in remediating technical debt issues identified by static code analysis and security scanning tools in software systems, in terms of resolution rate and accuracy?

The scope of this study is constrained by the chosen methodology and context. This research specifically focuses on Flavoria's system as a case study and thus limits the generalization of findings to the software systems with the similar characteristics. Though the analysis in this study explore the key development practices that contribute to technical debt, but it does not cover broader organizational and cultural influences. This thesis leaves the opportunities for conducting the future research to expand the findings to larger systems and validate the long-term impact of the usage of modern AI-driven technical debt assessment and management tools.

1.5 Contribution

This thesis makes the following contributions:

1. It investigates common development practices that lead to the accumulation of technical debt, using both literature and a real-world case study.

2. It evaluates how modern code analysis tools (SonarQube and Snyk) can support technical debt assessment and offer insight into code maintainability and security.

3. It provides a structured methodology to apply and assess the remediation capabilities of LLM-powered agentic AI (Claude Sonnet 4 via Cursor Pro) on a real-world codebase.

4. It offers a critical analysis of remediation outcomes, including resolution rates, false positives, and quality metrics, to determine the effectiveness of AI-driven maintainability strategies.

1.6 Thesis Structure

The thesis is structured as follows: Chapter 2 introduces concept of technical debt, its classification, types, causes and its impacts. It also discusses AI's role in software engineering and identifies research gap that this thesis aims to address. Chapter 3 presents modern available tools that can be used to identify and assess technical debt in a software system. Chapter 4 describes the methodology of experimental design, including the selection of the MyFlavoria codebase, data collection process, issues extraction from tools, and the agentic AI-based remediation process using Cursor Pro with Claude Sonnet 4. Chapter 5 reports the outcomes of the remediation experiment, including quantitative metrics such as issue resolution rate and insights about false positives and code quality improvements. Chapter 6 interprets

the results in the context of technical debt remediation strategies while addressing the research questions, identifies study limitations, discusses the implications for sustainable software maintenance and proposes directions for future research in AI-driven technical debt management. Chapter 7 summarizes the main findings and contributions of the thesis, outlining how the proposed approach supports technical debt management and its relevance to both research and practice. This is followed by an appendix that includes the source code used during the process.

1.7 Declaration on the Use of Generative AI

I hereby declare that I have used ChatGPT by OpenAI solely for the purpose of sentence structure refinement and grammar correction to enhance the clarity and readability of the text. All research design, implementation, analysis and results presented are entirely my own.

2 Background and Related Work

2.1 Overview of Technical Debt

Technical debt is a metaphor term which was first coined by Ward Cunningham in his technical report about the WyCash portfolio management system, where he stated that "shipping first time is like going into debt. The danger occurs when the debt is not paid. Every minute spent on not-quite-right code counts as interest on that debt" [7]. To introduce this concept, Ward Cunningham actually drawn a comparison between writing immature code and making an inflexible product, resulting in technical debt. In his report, he mentioned that writing immature or incomplete code may adequately perform well and meet customer expectations in the short-term, but having more of it could make the product unmanageable, which eventually results in a rigid product. He wanted to draw attention to the sensitivity of this matter to the software engineering organizations, and that's why the "debt" word came in to do this job.

Over time, this metaphor has evolved, and many modern perspectives on the technical debt has been developed by the collaborative efforts of the software engineering community. One notable example is the Dagstuhl Seminar 16162, titled *Managing Technical Debt in Software Engineering* [8], where researchers and practitioners gathered together to define and conceptualize the technical debt. According to the report from the seminar, technical debt is defined as: "In software-intensive

systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability" [8].

2.1.1 Classification of Technical Debt

After Ward Cunningham introduced the term "technical debt", it has become a widely researched topic [9], [10]. For classification of technical debt, many experts deep dived to understand this phenomenon and to create a classification framework. Prominent researchers like Steve McConnell and Martin Fowler started analyzing technical debt and categorized it's types by examining the reasons behind its creation. Another researcher, Philippe Kruchten, took a different approach by classifying technical debt types based on their level of detail and complexity.

2.1.1.1 McConnell's Debt Classification

Steve McConnell published a white paper on Managing Technical Debt which offers a comprehensive exploration of the technical debt, where he categorized it into two primary types: unintentional and intentional debt [11].

Unintentional debt is a debt which incurs due to low quality work, poor development practices, errors or lack of expertise, as a result of non-strategic decisions. For example, a junior software engineer may write inefficient or unstructured code because of limited experience, which creates a system that is harder to maintain or extend further. On the other hand, **Intentional Debt** is the debt which arises intentionally as a result of strategic decisions to meet immediate goals, such as shipping a product faster or saving the costs. For example, a startup which is building a MVP (Minimum Viable Product) may skip writing unit tests to launch the product

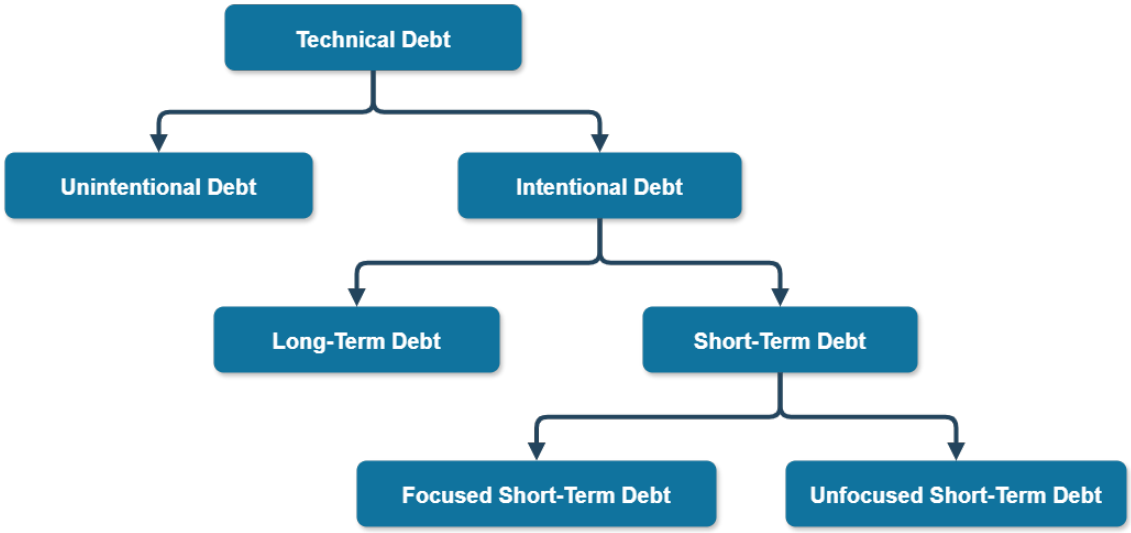

Figure 2.1: Steve McConnell's Technical Debt Classification [11]

quickly. The team knows that this decision could lead to maintenance issues in future if not fixed, but they somehow tends to prioritize their strategic decisions to get the product into the market as soon as possible, either to gain interaction or secure any funding. This further categorized into **short-term debt**, typically incurred tactically to meet earlier deadlines but often planned to be addressed in the next release cycle or soon after, and **long-term debt**, which is adopted strategically to achieve broader goals over extended timeline of few years or longer. Furthermore, short-term debt is divided into **focused** and **unfocused** types. **Focused Short-Term Debt** is due to well defined and specific shortcuts that are easy to track and manage. For example, during a critical/urgent release, a company might miss adding the internalization support in the product, which is easily track-able and can be addressed in the next release. While **Unfocused Short-Term Debt** arises due to minor inefficiencies such as skipping code reviews, inconsistent naming conventions or ignoring best practices under urgent deadlines. These small actions accumulate into massive technical debt that is difficult to identify and fix over time.

McConnell suggested that by focusing on structured decision making about how

and when to incur and address technical debt, and transparent debt tracking either by adding the debt as issues in the defect tracking system with required effort and schedule, or even by maintaining the list of technical debt issues as part of the Scrum backlog, can help companies to reduce the debt across product release cycles.

2.1.1.2 Fowler's Debt Perspective

Martin Fowler's expands the concept of technical debt by introducing a technical debt quadrant [12] as shown in Figure 2.2. Fowler classifies the debt based on the intent: Deliberate and Inadvertent, and context: Reckless and Prudent. This framework helps in understanding the nature and implications of different types of technical debt.

Reckless-Deliberate debt refers to the intentional shortcuts that teams knowingly take in a fully aware situation that they are compromising the quality of the code for speed without regard for long-term consequences. Developers choose "quick and dirty" solution to meet a tight deadline, for example, skipping automated testing to accelerate the feature delivery despite awareness of associated risks. This debt often incurs in toxic culture organizations where due to scheduling pressure, speed is prioritized over quality, which leads to the long-term system fragility [13].

Reckless-Inadvertent debt is accumulated due to the negligence or lack of awareness, without realizing the long-term consequences. For example, a messy code produced by a junior developer without mentor-ship which is then complimented by no code reviews.

Prudent-Deliberate debt is incurred consciously by teams to meet immediate goals, understanding the trade-offs, but also with explicit plans for repayment in the next sprint of the product release. For example, a startup implemented hard-coded values to release their MVP to meet investors deadline, with clean plan for refactoring those in next release. This is a strategic move when time to market is

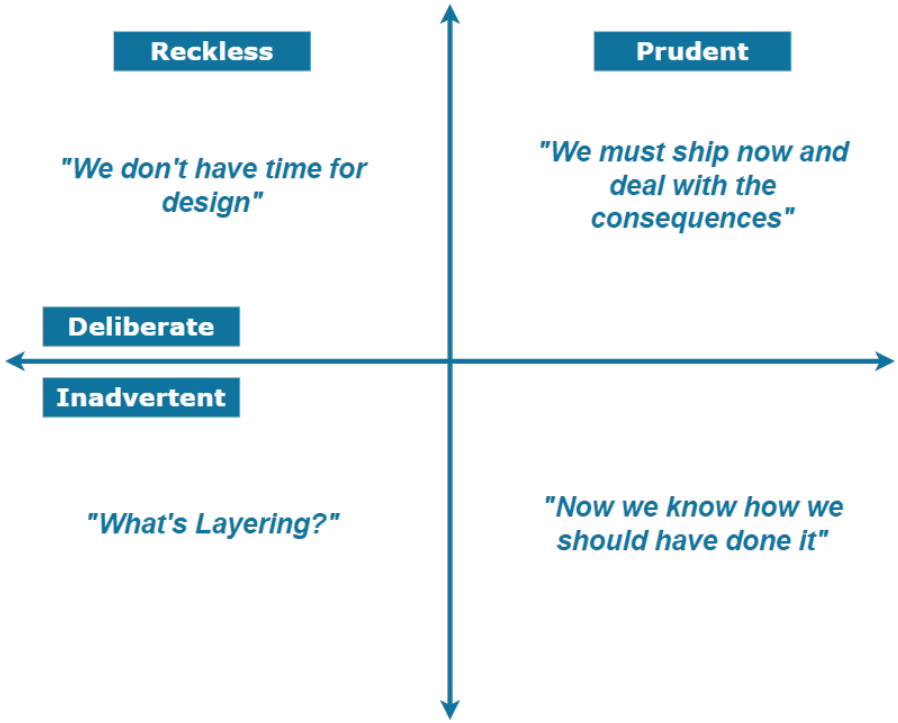

Figure 2.2: Fowler's Technical Debt Quadrant [12]

essential [13].

Prudent-Inadvertent debt arises when teams despite their best efforts, discover better design practices after implementation due to unforeseen gaps in their knowledge. For example, a team using monolithic architecture due to inexperience with micro-frontend architecture, later realizes that the initial design could be improved. This kind of debt is common and often inevitable because understanding deepens over time.

2.1.1.3 Kruchten's Granularity Analysis

Philippe Kruchten introduces the technical debt landscape which is a framework that broadens the understanding of technical debt by categorizing it based on visibility and granularity [13]. He distinguished technical debt from defects or unimplemented features/requirements, by focusing on aspects that are primarily invisible to end-users but significantly impact the development team's efficiency and the system's

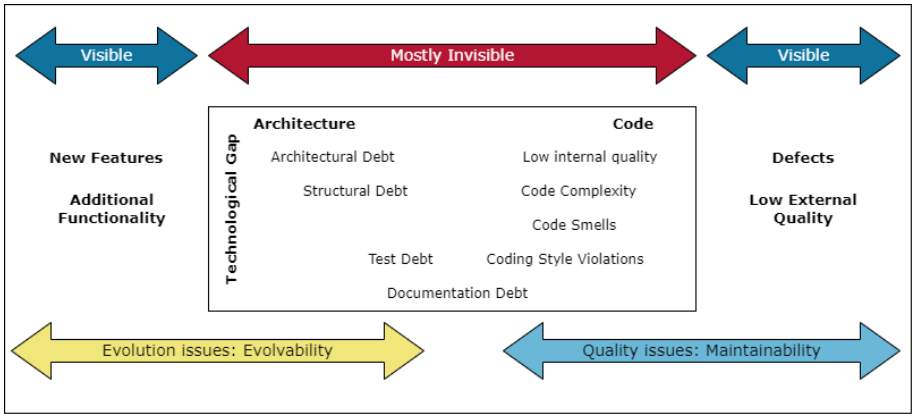

Figure 2.3: Kruchten's Technical Debt Landscape [13]

maintainability.

As shown in Figure 2.3, visible debt includes issues that apparent to customers or users, such as defects or missing features. However, Kruchten emphasized that these should not be classified as technical debt, as doing this would dilute the actual metaphor. While invisible debt includes internal issues that are not immediately evident to users but actually affects the development processes, such as convoluted code structures or architectural debt.

Kruchten further classified the invisible debt based on the granularity level: code-level debt and architectural debt. Code-level debt refers to the fine-grained issues within the source code of a software, such as violation of coding standards etc. This debt deteriorates the system's maintainability which makes future modifications more difficult. But with the help of various code analyzer tools [14], [15], these issues now can be identified and addressed more effectively. While architectural debt refers to coarse-grained issues related to design decisions such as the selection of technologies, platforms or architectural patterns. Accumulating this level of debt hinders the system's evolvability, which makes it difficult to adapt to new requirements and integrate emerging technologies. Unlike code-level debt, architectural debt is less detectable by automated tools and often requires thorough reviews to identify and mitigate.

Kruchten also highlighted that some debt arises due to changes in external environment such as technological advancements or shift in market demands. For instance, a system built on a technology stack that becomes obsolete incurs technical debt, as continuing with the outdated technologies may hinder future development and thus scalability of the software. He refers this to a phenomenon as "technological gaps". He emphasized that debt which incurs due to this phenomenon, requires more forward thinking and attention to mitigate because cost associated to these forms of debt tend to be significantly higher than those of code-level debts.

2.1.2 Types of Technical Debt

While Cunningham primarily focused on technical debt within the context of the code, the modern understanding acknowledges that technical debt can be seen across all aspects of software development [16]. So, understanding the types of the technical debt is essential to manage and mitigate its impact. The initial version of the types of technical was proposed in detail by Alves et al. [17] based on the studies until 2013. According to their studies, Technical debt can be categorized into many types which are presented below:

1. **Code Debt:** Refers to problems that arise when developers write messy code which is hard to maintain and affects its readability, for example by violating the coding standards or best practices. This is not about the functionality, because the code works, but it so difficult and confusing to make changes in the future.

2. **Architecture Debt:** Refers to issues that occurs in the product's architecture, or when the initial fundamental structure of the product is compromised, for example by violating the modularity rules that affects the performance or reliability of the product. Normally this is not something that can be fixed

with minor modification or with quick code patch, instead this requires a significant and extensive development work.

3. **Design Debt:** Refers to debt which occurs when developers take shortcuts that violates the good object oriented principles. For example big or tightly coupled classes where changing one thing could breaks another.

4. **Documentation Debt:** Refers to debt which builds up when the software project docs are missing, unclear or outdated. This occurs when developers do not adequately document their work, and it makes very difficult for other developers to understand the project.

5. **Test Debt:** Refers to debt which accumulates when the planned tests are skipped during the testing activities or the test suite coverage is very low. This debt occurs when teams in the organization prioritize their time in maintaining the existing tests instead of generating new ones to ensure that the software product is functional and bug-free.

6. **Defect Debt:** Defects that are known and are identified during the testing activities or by the users, and are logged in the bug tracking system, referred to Defect debt. These are the known bugs that the teams have decided to fix them later, but due to resource constraints or other high level priorities, these defects are pushed back and delayed which results in the accumulation of this debt and makes it difficult to overcome these issues later.

7. **Infrastructure Debt:** Refers to the debt which builds up when necessary fixes or updates are delayed in the infrastructure. These infrastructure problems in the software organizations obstructs the development activities which results in limiting the team capabilities to build a high quality product output.

8. **Requirements Debt:** This debt happens when team takes shortcut in feature

implementation. This refers to compromises which a development team make on what and how to implement. For example, you maybe implement only a part of the feature, or it is developed to work for some of the cases but not for all, or perhaps you completed the basic requirements of the feature but did not take into account the requirements which are non-functional such as security or performance etc.

9. **Versioning Debt:** Refers to the debt which occurs when the source code versioning of the software gets messy for example due to the creation of unnecessary code forks that now need to be maintained separately.

10. **Usability Debt:** This debt occurs when the teams make poor interface decisions that needs to be fixed later. This can be for example due to ignoring the basic usability standards or by having inconsistencies in the user interface.

11. **People Debt:** Refers to debt which occurs when the team's knowledge or skills are not properly distributed. For example, when expertise and knowledge is limited to just few individuals due to delayed hiring or training, it can slow down or block certain parts of the development.

12. **Test Automation Debt:** Refers to debt which accumulates when teams keep putting on automating the tests for the existing features to move forward to support the continuous integration.

13. **Process Debt:** This debt happens when teams keep using or following the outdated or inefficient processes just because "that's how we have always done it", or the process that was designed for something might be no longer applicable. For example, it's like still using the paper forms in this digital era.

14. **Build Debt:** This debt is what happens when the build process becomes a mess. Maybe you have got the unnecessary code there which is not contribut-

ing to any of the functionalities that provide some value to the user, or the dependencies are so tangled that builds take forever. This makes the build process unnecessarily slow.

15. **Service Debt:** This debt is particularly relevant for systems using web services. It occurs when you pick the inappropriate or wrong service for your needs which leads to either waste (like paying for the resources that are not required or that you don't need) or poor performance. This debt is an example of the mismatch between the software application requirements and the service features.

2.1.3 Causes and Impact on Software Lifecycle

The accumulation of technical debt is a prevalent challenge in modern software development which emerges as a result of intentional trade-offs and unintentional negligence made under constraints such as tight deadlines, evolving deadlines or legacy practices. It originates from combination of human-related and environmental factors. According to the InsighTD global survey, 653 practitioners from 6 countries identified **deadlines**, as shown in Figure 2.4, as the single most cited cause of technical debt [18][19], where Agile teams often prioritize rapid MVP delivery over sustainable code quality, with lack of knowledge, inappropriate planning and lack of qualified professionals as other most cited causes after deadlines. While on the other hand, **delivery delay**, as shown in Figure 2.5, is the most cited effect of technical debt, following low maintainability and rework, among others.

Technical debt in the software lifecycle arises also due to business pressures and managerial decisions [20]. For example, sometimes companies have limited funds to implement all required practicalities but the market wants new features or urgent modifications, which forces them to prioritize features over code quality [20]. In support of this, an empirical study on technical debt in Finnish SMEs revealed

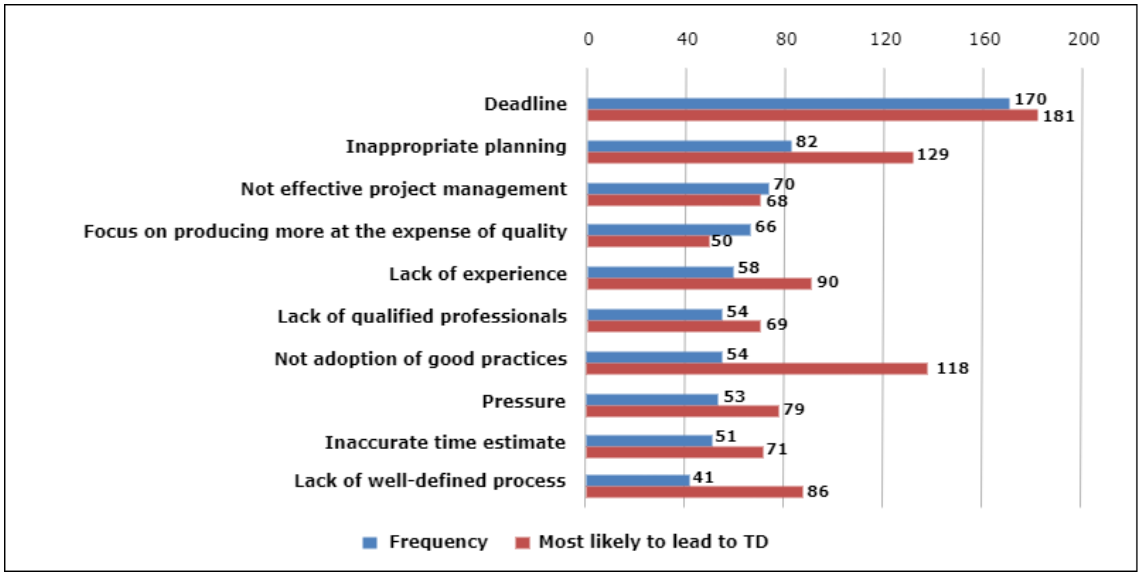

Figure 2.4: 10 most frequently reported causes of TD [19]

that the tight budgets, time constraints, and challenges with effort estimation are the top reasons that frequently contribute to the technical debt accumulation [21]. The study revealed that the problems such as insufficient test automation as well as inadequate early validation of requirements can result in high levels of debt in testing and requirements. The authors suggest that early and clear communication with stakeholders and careful estimations are the necessary measures to avoid these issues because these strategies helps in learning from the customer feedback and improving the estimation processes. Another cause of incurring technical debt is the urgency to release features before competitors, which drives rushed development, which in turn leads to tight deadlines [20]. It has also seen that accelerating deployments to get competitive advantages over others also incurs technical debt which significantly degrades system's performance over the software life cycle, with 10% increase in debt, reducing gross profitability by 16%, which is a sign of reduction of long-term business value [22]. However, with the presence of experienced IT teams and chief information officer (CIO), these long-term impacts can be partially mitigated [22].

Technical debt, specifically code debt, occurs when software is developed without

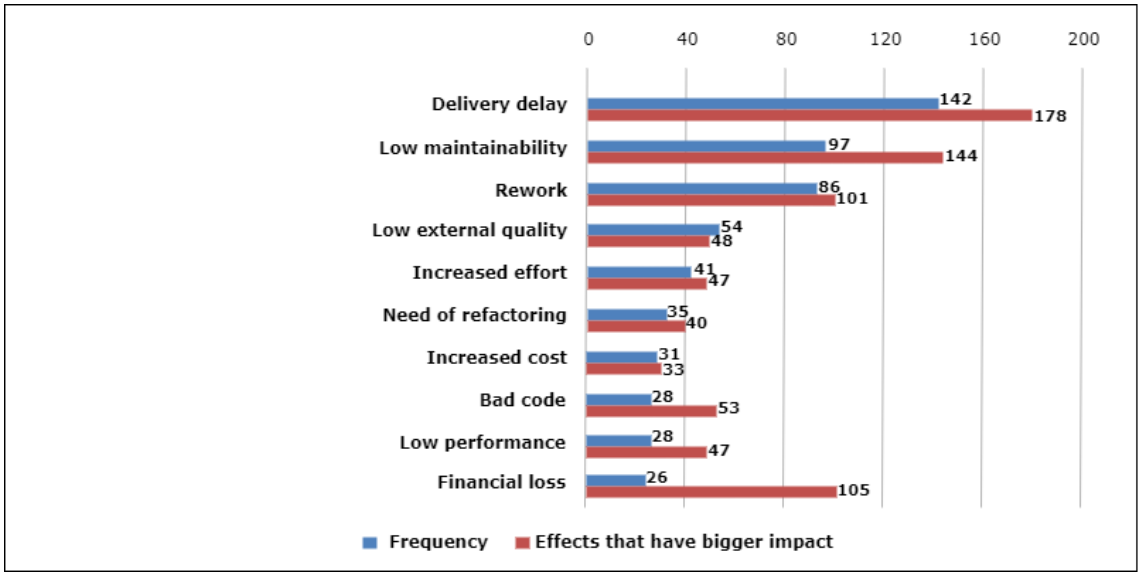

Figure 2.5: 10 most frequently discussed consequences of TD [19]

maintaining or following best coding practices. This debt can build up to the point where the software becomes difficult to maintain, potentially leading to technical bankruptcy [23]. And due to this, development process of a software product can be obstructed and according to Besker et al. [1], 23% of total developer's time is wasted due to experiencing the technical debt in the software development life cycle. According to a research by IBM System Science Institute to find out the relative cost of fixing code level debt in the SDLC process stages, they found that defects found in maintenance phase were almost 15 times more costly to fix than in implementation phase [24]. Figure 2.6 depicts the results.

Technical debt is becoming an increasingly important challenge for businesses that want to find a balance between innovation and maintainability. A recent report in 2024, published by Accenture [2], based on the results of global survey of 1500 companies and extensive industry research, provides more in-depth analysis of the growing impact of the technical debt, particularly in the age of AI. According to the report, in the US alone, technical debt costs approximately US$2.41 trillion annually and would require an estimated US$1.52 trillion to remediate this debt. And the

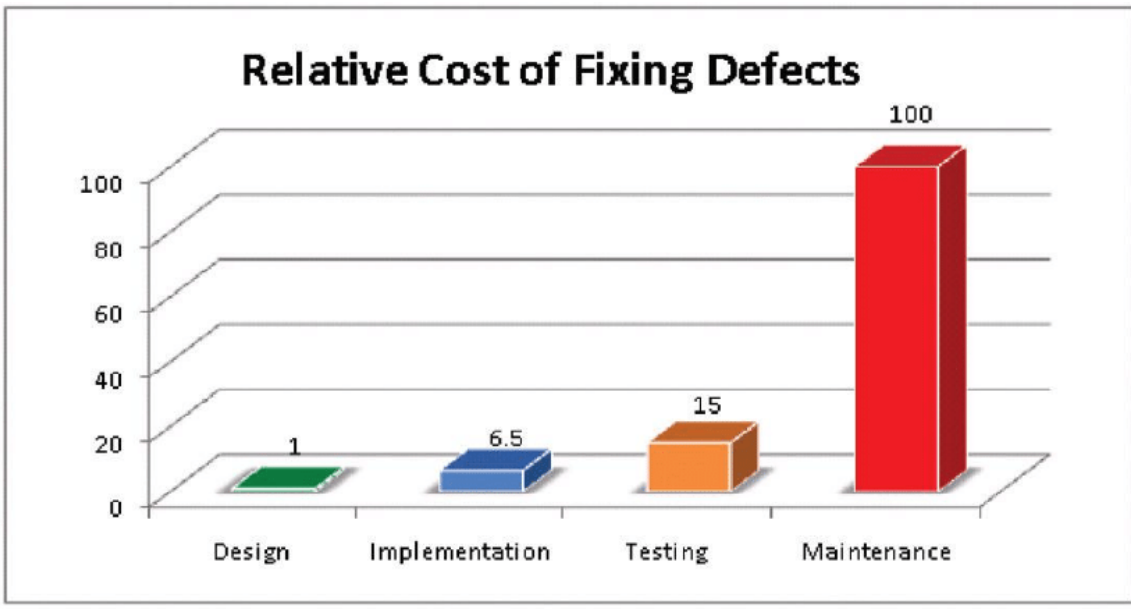

Figure 2.6: IBM Study: Defect Fixing Costs Throughout the SDLC [24]

factors contributing to this amount of technical debt includes legacy code, outdate frameworks, and technologies specifically designed for human interactions.

As companies adopt always-on agile IT solutions [2], maintaining a balance between speed-to-market and development of maintainable software systems is still a challenging task. Due to organizational pressure and shifting market conditions, IT leaders tend to prioritize immediate business needs which often results in postponing essential maintenance or refactoring efforts, which further increases technical debt over time [2]. The accumulated debt not only slows down feature development and bug fixes but also drives up the maintenance costs. An important insight from the report [2] highlights that from 2023 onward, executives of many companies plan to increase their technical debt remediation budgets by about 7 percent: from 17% to 24% of IT budgets.

Architectural Technical debt often stems from deprecated frameworks, and tightly coupled components in monolithic systems increase maintenance costs and lower stability, as highlighted in this study [25]. Also, when components in a software

becomes too tightly coupled, it becomes too challenging to isolate and maintain individual modules [25]. Nilsson et al. [26] suggest that in the pre-deployment phase which is basically the development phase of a software product, it is important to avoid Architectural technical debt. However, they also acknowledged that the other concrete debts such as test, code, or documentation debt, can be incurred during the development phase and those can be acceptable. They emphasized that the effective communication early in the development process is crucial to prevent decisions that could lead to future technical debt.

In software maintenance, one of the key challenges is managing the external dependencies like third-party libraries and frameworks integrated into the software. According to software engineering experts, regularly updating the software dependencies is crucial for ensuring long-term maintainability of the software system to reduce technical debt in time [27]. Significant effort along with extra sense of responsibility is required to achieve this, but many developers comply to the practice "if it ain't broke, don't fix it", and majority of software are using outdated dependencies [28]. Keeping your project's dependencies up to date is really important in today's software development, but it is also a bit challenging at the same time. Projects can rely on dozens or even thousands of libraries, and updating a dependency might introduce changes that break things, which can be really hard to detect and fix [29]. For example, a study by Kula et al. [28] found that 81.5% of the 4600 Java/Maven projects on Github still rely on outdated dependencies, mainly because developers either are not aware of updates or find the extra work too overwhelming. Similarly, Pashchenko et al. [30] discovered through interviews that developers often face tough choices when updating dependencies, having to balance issues like vulnerabilities, potential breaking changes, and internal policies.

2.2 Technical Debt Management Process

For the software maintenance, specifically in the effective management of technical debt, there are multiple steps involved as discussed by Li et al. [9]. The initial step is to identify problems within the software system, for example by using different techniques such as static code analysis. The next step is to log identified debt into a tracking system and categorize debt based on type, severity and potential impact. Once documentation of debt is done, quantifying the impact using for example maintainability score and estimated effort(cost) to fix, is the next activity. Then, in TD prioritization, identified debt items are ranked based on business impact, technical feasibility and urgency, to decide which debts could be addressed in upcoming sprints for example. The next action is to inform stakeholders about the risks and justify technical debt repayment in terms of business. Then, in TD monitoring phase, TD is continuously tracked using dashboards, automated tools and through sprint reviews. The next phase is the TD repayment, where sprint time is specifically allocated to resolve high-priority debt such as by refactoring, upgrading dependencies, improving tests etc. At last, best practices are implemented such as automated testing, CI/CD pipelines to prevent TD accumulation.

Likewise, other studies also support this set of activities for effective technical debt management in software systems [31]. TD repayment, TD identification as well as TD measurement are the most discussed activities in most of the studies [9], [31], and these activities involve too much of manual work for example for identifying code or architectural debt, or outdated and vulnerable dependencies, which is a very time consuming task. This indicates the significance of utilizing automated tools to carry out these activities. For this reason, Chapter 3 will explore various available tools.

2.3 AI Applications in Software Engineering

In the realm of modern software development, Artificial Intelligence has emerged as a state-of-the-art to revolutionize the industry. AI based tools have been increasingly being used to address and solve critical challenges such as optimizing code quality, identifying and resolving technical debt, and automating the repetitive maintenance tasks [32]. A comprehensive literature review analyzed 15 research papers on AI-powered tools for Technical debt management and the results highlight the emergence and significance of AI techniques integration in different phases of technical debt management process [32].

Recent progress in Artificial Intelligence, especially with large language models (LLMs) and agentic AI systems, has created new opportunities for tackling technical debt. Large language models (LLMs) power agentic AI systems that autonomously perform coding tasks, which can offer significant potential for managing technical debt. This chapter provides a background on AI-driven approaches to technical debt management by synthesizing related researches. It highlights the evolution of technical debt management, the role of AI and the research gap that thesis aim to address.

2.3.1 Evolution of Technical Debt Management

As early researches focused on classifying technical debt into categories such as code, design and documentation debt [10], [13], tools like SonarQube [33] has emerged to detect technical debt through static code analysis, identifying issues like code smells and complexity metrics [9], [14].

Machine learning has also introduced the automated technical debt detection methods. Integrating AI-based techniques into the code reviews has also shown promising results in enhancing software quality and efficiency. A study introduced AICodeReview [34], an IntelliJ IDEA plugin that utilized GPT-3.5 to automate

code assessments. It effectively detects both syntax and semantic errors while suggesting possible fixes. Early results showed that this tool significantly cuts down review times and enhances the detection and refactoring of code smells compared to manual reviews. In another effort, Li et al. [35] created CodeReviewer, which is a pre-trained model designed to automate code review tasks by analyzing large datasets of code changes and reviews from multiple programming languages. This model has shown superior performance in estimating code quality, generating review comments, and refining code. Moreover, DebtViz [36] was introduced as an AI-powered tool that automatically detects and classifies Self-Admitted Technical Debt (SATD) by analyzing source code comments and issue trackers using Convolutional Neural Network, by providing real-time insights into technical debt. Similarly, Mangave et al. [37] introduced an automated code review tool that aims to improve the efficiency and effectiveness of code assessments. The development of these kind of AI powered tools demonstrates a growing trend in integrating artificial intelligence not only to ease the code review process but also to manage technical debt effectively in software development. Another study explored natural language processing (NLP) methods, such as BERT models, by analyzing code comments and commit messages to find out self-admitted technical debt (SATD) [38].

However the remediation process remains mostly manual, depending on refactoring and code reviews, which can be slow and labor-intensive, and prone to mistakes, particularly in large and complex codebases. Even though the above mentioned approaches improved detection accuracy but left remediation to developers. A systematic evaluation of 29 tools and their support for technical debt activities reveals that, only 1 out of 29, facilitates technical debt repayment, highlighting a significant lack of tools and techniques in this area [9].

2.3.2 Role of AI and LLMs

Large language models like GPT and BERT have transformed software engineering by simplifying tasks such as bug detection, code generation, as well as documentation [39]. As LLMs are trained on the vast amount of datasets of code and text, they can understand programming languages and generate contextually relevant solutions. For example, a study investigated OpenAI's Codex which is a GPT-3-like model, which has effectively identified and fixed bugs in Python and Java [3]. Another research indicates that large language models such as codex and GPT-3 can effectively generate repair patches for JavaScript code issues [4]. A recent study also show that ChatGPT can effectively support the automated program repair and has better performance over other deep learning models [5]. These results indicate the capability of LLMs to reduce technical debt by generating bug fixes which enhances software maintainability.

Agentic AI, which is a subset of AI, is capable of autonomous planning, reasoning and tool usage, and it represents a significance advancement in this modern era. Studies like [40] and [41] describe agentic LLMs that perform complex workflows, such as multi-file code editing or executing development commands. Industry tools like Windsurf (formerly Codeium) [42] and Cursor [43] leverage these capabilities and offer autonomous code editing and deep codebase reasoning. These developments indicates that agentic AI can transform technical debt management by automating detection and remediation.

2.3.3 AI-Driven Technical Debt Remediation

Recently, studies have started to examine the capabilities of AI in terms of technical debt management, mostly looking at how AI can help in debt detection. A systematic literature review analyzed several NLP-based methods and found that models like BERT are good at identifying SATD, but didn't explore the ways to address

remediation [44].

Microsoft's CORE tool is interesting becuase it uses two LLMs together to create and rank code updates for the issues identified by tools like SonarQube and CodeQL [45]. By using CORE, reductions in human review effort were achieved, since 59.2% of Python files and 76.8% of Java files were successfully revised. But it only covers problems in the code quality specifically like bugs and style mistakes, rather than technical debt, and it's LLMs also lack the autonomy of agentic AI systems.

Tools like Cline, Tabby, Cursor, Windsurf, Codiga, Intellicode utilize AI to examine code and discover vulnerabilities, and then suggest code improvements and fixes like how to fix them [46]. But these tools still lack the evaluation and testing to show how effectively these tools can address technical debt remediation.

2.3.4 Research Gap and Motivation

Most studies focus on technical debt detection and does not address it's repayment, leaving this hectic work left to the developers. Researches like [44] mostly examine SATD instead of broader technical debt issues identified by static code analyzers and security scanners. Even though agentic AI tools were discussed in [47], but the use of this technology for resolving technical debt is still not explored. Also, industry tools like Cline, Windsurf, Cursor showcase practical applications, but there is a lack of evaluation of measuring the effectiveness of such tools in repaying the technical debt issues identified by different tools to mitigate technical debt and enhance software maintainability.

This thesis addresses these gaps by evaluating the effectiveness of LLM-powered agentic AI in remediating the technical debt issues identified by static code analyzers and security tools. Unlike CORE, which only focuses on general code quality issues identified by SonarQube, this thesis will utilize the capabilities of agentic AI (Cursor) to address technical debt specifically. By providing empirical evaluation and

probably by developing a new proof of concept framework strategy, this thesis will showcase that how AI-driven maintainability strategies can be utilized for technical debt management.

3 Tools for Technical Debt Management

3.1 Classification and Role of Tools

To mitigate the maintainability issues early on in the SDLC, several tools have emerged as powerful assets in the technical debt management for software maintenance [6], [9], [31], [48], specifically for code review and dependency management. And, selecting the right tool to utilize for a specific software project is a prevalent challenge [48]. Recent research [49] shows that most technical debt management tools concentrate on three areas: source code issues, architectural problems, and design flaws. The consistent focus these areas suggests that they are major challenges for developers, as they greatly impact software maintainability, performance and scalability.

3.2 Automated Code Review

Traditionally, code review involves developers manually analyzing and scrutinizing the code to locate potential issues such as bugs, violation of coding standards as well as inefficiencies. Several tools are available and currently being used to automate this part of the process such as SonarQube, DeepCode, and CAST etc.

SonarQube, an open-source static code analysis tool, offers automated code re-

views, focusing on code quality and maintainability metrics of a software, and it supports multiple programming languages. It analyzes the code comments, whole source code and commit messages to identify and measure the technical debt severity, and it offers beneficial insights to adhere to best coding standards to enhance the long-term software maintainability. Table 3.1 and Table 3.2 presents the list of available tools for code review in terms of what type of technical debt they identify, what is their technical debt estimation strategy, which platforms and languages they support [48], [50].

Table 3.1: Automated Code Review Tools

Name (Release Year)	**Type**	**Technical Debt Estimation Formula**
CAST [51] (1998)	Architectural, design and code	violations × rule criticality × effort
NDepend [52] (2007)	Architectural, design and code	violations × fix effort
SonarQube [33] (2007)	Code	cost × nLOC
SQuORE [53] (2010)	Design and code	Not explicitly specified
Codacy [54] (2016)	Code	Not explicitly specified
Designite [55] (2016)	Design and code	Design rules violated
Code Inspector [56] (2019)	Architectural, design and code	function of violations, duplications
SymfonyInsight [57] (2019)	Code	number of issues × time needed

Table 3.2: Additional Features of Automated Code Review Tools

Name	**Platform**	**Integration**	**Output**	**Languages**
CAST	Windows	Jenkins and Maven	API and GUI	Most
NDepend	Windows	Azure, Jenkins and VS	GUI	.Net frameworks
SonarQube	Independent	Eclipse, IntelliJ, and VS	All*	Most, with plugins
SQuORE	Independent	No	API and GUI	C++, Java, others with plugins
Codacy	Independent	GitHub, GitLab, Bitbucket, CLI, API, CI/CD	API and GUI	Most, over 40 languages
Designite	Independent	VS, IntelliJ	GUI	Java, C#
Code Inspector	Independent	GitHub, Bitbucket, GitLab, Jenkins and Travis	API	Most
SymfonyInsight	Independent	No	GUI and CI	PHP

3.3 Dependency Management

Although dependency-related technical debt is not directly addressed in this thesis, but this section presents overview of tools that are used in the industry for proactive dependency management in reducing architectural and versioning debt [28], [58].

Several dependency management tools such as Dependabot [59], Renovatebot [60], PyUp [61] and Snyk-bot [62] have been adopted and being used by industry teams at large scale and these tools are responsible of creation of millions of pull requests on Github [63]. PyUp rebranded to Safety Cybersecurity in July 2023 [64], expanding its focus beyond Python to include ecosystems like JavaScript, Java, .NET, Go, and Ruby. These tools automatically monitor the code bases and identify vulnerable and outdated dependencies, and then open Pull Requests (PRs) with the required changes based on the stability, and compatibility with the existing code base. These tools can reduce the accumulation of technical debt due to obsolete libraries/dependencies, and automate the dependencies update process to improve the longevity of the software.

Dependabot [59] stands out as the most widely adopted dependency management bot across GitHub repositories [63], [65]. It was first introduced in 2017 as Dependabot Preview, and later acquired by GitHub in 2019 [66]. In August 2021, the preview version was discontinued in favor of a fully integrated GitHub-native version launched in June 2020 [67]. This updated tool provides two main capabilities: version updates, which use a dependabot.yml configuration file to automatically create pull requests for keeping dependencies current; and security updates, which scan GitHub repositories for known vulnerabilities and notify maintainers, even in the absence of a configuration file. Alfadel et al. [58] conducted a study on JavaScript open-source projects using Dependabot. They found that 65.42% of security-related pull requests created by Dependabot were accepted by developers and most of these accepted pull requests were merged within a day, which indicates a swift response to security updates. Also, according to another study conducted by Runzhi he et al. [68], projects reduced their technical lag from an average of 48.99 days to 25.38 days after using the Dependabot for 90 days. Additionally, 35.7% of projects reached zero technical lag within this time. This indicates the importance of these tools

in notifying updates to fix dependency issues as soon as possible to enhance the productivity of the teams. Table 3.3 gives overview of the available tools and what languages they support and how do they work.

Table 3.3: Dependency Management Tools

Name (Release Year)	**Ecosystem**	**Debt Mitigation Strategy**
Dependabot (2017)	Multi-language	Automates updates and alerts for outdated dependencies.
Renovate (2017)	Multi-language	Auto-updates dependencies to reduce maintenance overhead.
Snyk (2015)	Multi-language	Flags and fixes vulnerable dependencies.
Sonatype Nexus Lifecycle (2016)	Enterprise (Java, .NET, Node.js)	Assesses risk and guides safe component use.
OWASP Dependency-Check (2011)	Multi-language	Detects known vulnerabilities in dependencies.
WhiteSource (2013)	Multi-language	Ensures security and license compliance.
Safety (formerly PyUp) (2016)	Multi-Language (Python, Java, Javascript, Go, .NET and Ruby)	Automates dependency updates via pull requests.

4 Methodology

This chapter outlines the methodology for evaluating the effectiveness of large language models (LLM)-powered agentic AI, specifically the Cursor [43] Pro version with the Anthropic's Claude Sonnet 4 [69] which is the latest released advanced model, in remediating technical debt issues identified by static code analysis and security tools. This work addresses the gap highlighted in the existing literature, which lacks systematic evaluations of autonomous AI systems for technical debt remediation. The evaluation is focused exclusively on the MyFlavoria PWA [70], an Ionic-based React application which is designed and developed for the users of Flavoria's restaurant in Turku, Finland [71]. This methodology is designed to be reproducible, using accessible tools like SonarQube and Snyk, and provides actionable insights for improving the MyFlavoria's maintainability, aligning with the researcher's involvement in the project.

4.1 Research Design

This study employed an experimental design structured around a diagnostic-proactive approach for managing technical debt. The diagnostic approach focuses on identifying technical debt issues using static code analysis and security tools, while the proactive approach leverages agentic AI powered by large language models to remediate the identified issues. This approach is composed of three phases: (1) identification of technical debt using static code analysis and security tools, (2) application

of Cursor Pro with LLM model to remediate identified issues, and (3) evaluation of remediation outcomes using quantitative and qualitative metrics.

Figure 4.1 illustrates the workflow, depicting the sequential steps across the three phases, from technical debt identification to final evaluation.

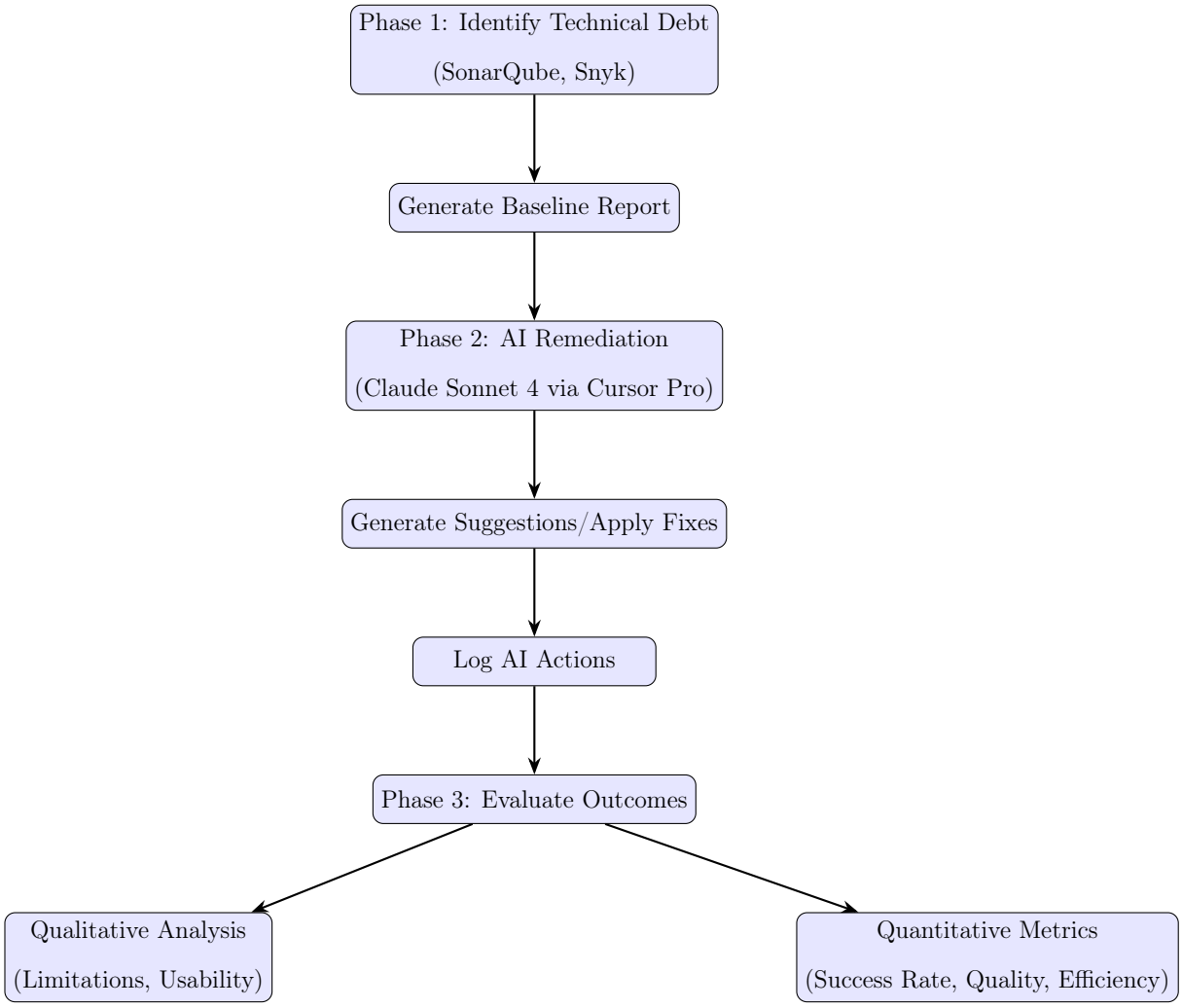

Figure 4.1: Flowchart of the experimental methodology for evaluating Claude Sonnet 4 via Cursor Pro in remediating technical debt.

In the first phase, technical debt is identified using two categories of tools:

- Static code analysis (SonarQube)

- Code security and vulnerabilities scanner (Snyk Code)

These tools are selected due to their comprehensive detection capabilities and strong industry adoption [15]. The output of this phase is a detailed combined list of issues of technical debt, which are categorized into code smells, bugs and vulnerabilities.

In the second phase, an agentic AI-assisted development tool (Cursor Pro with Claude Sonnet 4) is used to remediate those identified issues. The agentic AI model suggests and applies code changes autonomously based on the context and prompts derived from Phase 1 outputs.

Finally, the AI-generated remediations to those identified issues are evaluated by re-running the Phase 1. The new scan results are then compared with the baseline report to assess the effectiveness of the AI remediations, and perform analysis to get metrics such as resolution rate, false positive identifications and others. The remainder of this chapter explains the process of each phase in detail.

4.2 Data Collection

The data collection phase involved gathering technical debt issues data from a real-world functioning production-level codebase: MyFlavoria PWA. This phase covers the reasons behind the project selection, and the exact steps taken to collect and normalize the data from two industry standard tools: SonarQube and Snyk.

4.2.1 Codebase Selection

For the purpose of this study, the newly developed MyFlavoria PWA is selected as a primary codebase for technical debt analysis and tool-based evaluation. This app is developed to provide insights about the nutritional information to the users related to the food that they take with personal data tracking, and deployed as PWA as part of Flavoria's research platform at the University of Turku. This codebase was

chosen due to it's modern architecture, active development status, and researcher's involvement which enabled direct access and the opportunity to provide actionable insights for improving maintainability. Other reasons to chose MyFlavoria PWA codebase for this study are:

- It represented a major architectural decision of complete re-development to replace the legacy React Native implementation, which had accumulated significant technical debt due to outdated libraries and architecture.
- In opposite to legacy app, the new PWA app was developed using React framework, enhanced with Ionic-based UI components, and written in TypeScript, which enabled better scalablity and maintainability.
- MyFlavoria PWA was developed with a clear emphasis on maintainability best practices, including modular design, reusable components, and maintaining up-to-date dependencies.
- The structure and tooling of the PWA allow easy integration and testing of automated code analysis tools such as SonarQube and Snyk Code for security and vulnerabilities scanning.

4.2.2 Technical Debt Identification

This section presents the full technical process used to identify technical debt using two tools: SonarQube and Snyk, along with the data normalization, batching logic for AI input and internal data transformation scripts.

4.2.2.1 SonarQube Analysis

SonarQube (version v25.5.0.107428 Community Edition) was selected as the static code analysis tool to identify technical debt in the MyFlavoria Progressive Web App (PWA). Its selection was based on the following rationale:

- Proven static analysis capabilities for detecting code smells and maintainability issues [15].
- Comprehensive support for JavaScript and TypeScript codebases, aligning with MyFlavoria's tech stack.
- Customizable rule profiles, allowing tailored analysis for the project's needs.
- Free availability through the Community Edition, ensuring accessibility for academic research.

The setup of SonarQube followed a structured process to ensure accurate analysis of the MyFlavoria codebase, as detailed below:

1. **Pulling the Docker Image and Running SonarQube Locally**: SonarQube was deployed locally using Docker to facilitate analysis. The following command was executed to pull and run the SonarQube v25.5.0.107428 latest Community Edition container:

```
docker run -d --name sonarqube -p 9000:9000 sonarqube:
   latest
```

2. **Accessing the SonarQube Web Interface**: The web interface was accessed at `http://localhost:9000` to initialize the environment and create a new project for MyFlavoria.

3. **Installing SonarScanner CLI**: The SonarScanner command-line interface was installed globally using npm to enable code analysis:

```
npm install -g @sonar/scan
```

4. **Configuring the Project**: A configuration file, `sonar-project.properties`, was created in the root of the MyFlavoria project to define analysis parameters:

```
sonar.projectKey=MyFlavoria-Master
sonar.projectName=MyFlavoria Master
sonar.sources=.
sonar.host.url=http://localhost:9000
sonar.token=sqp_2d9526f4ffac3fdb3293d19a2f30583395fcd77d
```

5. **Running the Analysis**: The analysis was performed by executing the following command in the project root:

```
sonar
```

6. **Accessing the Report**: Results were accessed through the SonarQube web interface at `http://localhost:9000` and via the SonarQube API for programmatic extraction.

4.2.2.2 Extracting Issues Programmatically

Since the SonarQube Community Edition lacked built-in export features, a custom Python script was developed to retrieve issues programmatically via the SonarQube API. The script is provided in Appendix A (Listing A.1) and it used the following API request to fetch issues for the MyFlavoria project:

```
GET http://localhost:9000/api/issues/search?components=
   MyFlavoria-Master&issueStatuses=OPEN&ps=500
```

The retrieved issues were parsed into a structured JSON format for further processing, with each issue including details such as key, description of the issue, file, line number, severity, type, and suggestion(if available). All extracted issues were consolidated into a single JSON file for use in subsequent remediation steps with Cursor.

An example of the parsed output of single issue is shown below:

```
{
  "key": "e0730d6a-8c02-41d4-9784-5482bb54aee2",
  "description": "Make this public static property
     readonly.",
  "line": 8,
  "file": "src/API/BackendFactory.tsx",
  "severity": "MINOR",
  "type": "CODE_SMELL",
  "rule": "typescript:S1444",
  "suggestion": "Public \"static\" fields should be read-
     only"
},
```


4.2.2.3 Snyk CLI Analysis

Snyk CLI (version 1.1296.2) was selected as the security scanning tool to identify vulnerabilities in the MyFlavoria Progressive Web App (PWA) codebase. Its selection was based on the following rationale:

- Excellent vulnerability scanning using it's DeepCode AI and static analysis capabilities for modern JavaScript and TypeScript projects, aligning with MyFlavoria's tech stack.
- Support for CLI-based analysis, enabling integration into automated workflows.
- Direct JSON export functionality, facilitating programmatic processing of results.
- No requirement for access to proprietary code, ensuring compliance with ethical considerations for the MyFlavoria project.

The setup and execution of Snyk CLI followed a structured process to ensure comprehensive vulnerability detection, as detailed below:

1. **Installing Snyk Globally**: Snyk CLI was installed globally using npm to enable command-line access for scanning:

```
npm install -g snyk
```

2. **Authenticating via Web**: Authentication was performed to connect Snyk CLI to the Snyk platform, using the following command:

```
snyk auth
```

3. **Running Code Test and Exporting to JSON**: The MyFlavoria codebase was scanned for security issues, with results exported to a JSON file using the following command:

```
snyk code test --json > snyk_issues.json
```

   This command generated a structured JSON output containing identified vulnerabilities and insecure code patterns.

4.2.2.4 Data Transformation and Integration

Since the raw JSON output of the issues from Snyk was different in the format as compared to SonarQube, so it required to transform it into similar format. A custom Python script was developed to parse and normalize issues into a unified format. Each issue was transformed into a JSON object containing key fields such as key, description of the issue, file, line number, severity, type, and suggestion(if available).

An example of the normalized output is shown below:

```
{
```

```
      "key": "9aeeb692-cefa-4f93-8547-8176960e9b82",
      "description": "Avoid hardcoding values that are meant
         to be secret. Found a hardcoded string used in here."
         ,
      "line": 6,
      "file": "src/platform/abstraction.ts",
      "severity": "CRITICAL",
      "type": "VULNERABILITY",
      "rule": "javascript/HardcodedNonCryptoSecret",
      "suggestion": ""
}
```


Once the issues from both SonarQube and Snyk were transformed into unified similar format, they were then merged into a single master file, `combined_issues.json`, for use in subsequent remediation steps with Cursor using the Python script which is provided in Appendix A (Listing A.2).

4.2.2.5 JSON Chunking for AI Context Window

To accommodate the memory and token constraints of large language models (LLMs) like Claude Sonnet 4, used in Cursor, the `combined_issues.json` file was split into smaller chunks containing 10 issues each. A Python script was developed to perform this chunking, as shown below:

```
import json


def split_issues(input_file, chunk_size=10):
    with open(input_file, "r") as f:
        data = json.load(f)
        issues = data["issues"]
```

```
    for i in range(0, len(issues), chunk_size):
        chunk = issues[i:i+chunk_size]
        with open(f"issues_part_{i//chunk_size + 1}.json", "w"
           ) as out:
            json.dump({"issues": chunk}, out, indent=2)
```

Each chunk was semantically tagged with batch identifiers to facilitate tracking during remediation. These chunks were then used as input for Cursor to generate remediation suggestions.

4.3 AI-Driven Remediation

This section outlines the process used to remediate technical debt identified by SonarQube and Snyk in the MyFlavoria Progressive Web App (PWA) using Cursor Pro, an AI-enhanced development environment powered by Claude Sonnet 4. The approach was designed to evaluate the effectiveness, limitations, and practical integration of large language models (LLMs) into a real-world software maintenance workflow.

4.3.1 Tool Overview

Cursor Pro [43] is a modern code editor integrated with AI capabilities. It includes features such as context-aware autocompletion, multi-file deep codebase reasoning, and an agentic mode [72] that enables the AI to process structured inputs, perform reasoning across the codebase, and suggest or implement code changes.

Claude Sonnet 4 [69], developed by Anthropic, is a transformer-based LLM known for its high-quality code generation, reduced hallucination rate, and support for multi-modal and multi-file reasoning. It was selected for this experiment based on the following rationale:

- Its ability to interpret structured JSON inputs, critical for processing SonarQube and Snyk reports.
- A context length capability of approximately 200,000 tokens, sufficient for handling batched issue files.
- Its strong performance in previous academic and industrial coding benchmarks.

The integration of Claude Sonnet 4 into Cursor Pro enabled the development of a semi-automated pipeline for technical debt remediation.

4.3.2 Remediation Process

The AI-driven remediation was performed through an iterative, batch-based workflow using the JSON files generated from SonarQube and Snyk, each containing 10 technical debt issues. This segmentation of issues into chunks is done due to the context limitations of Claude Sonnet 4 while ensuring a focused and manageable scope for the AI.

1. **Issue File Ingestion**: Each batch file (e.g., `issues_part_1.json`) was attached to a Cursor Pro session. A structured prompt was written to guide the AI in reading, understanding, and acting upon the list of issues. The prompt template used is shown below:

```
You are a software engineer assistant tasked with repaying
    technical debt across an entire codebase.

You are given a file `issues_part_1.json`, which contains
   an array of issues identified by SonarQube and Snyk.
   Each issue has:
- `file`: path to the file
```

```
- `line`: line number of the issue
- `type`: CODE_SMELL, BUG, or VULNERABILITY
- `description`: a short explanation of the problem

Your task is as follows:

1. For **each issue**, navigate to the specified file and
   line number.
2. Use deep understanding of the codebase and context
   before taking action.
3. If the issue is valid:
   - Fix it without breaking app functionality.
4. If the issue is a **false positive**, **do not ignore
   it**. Instead:
   - Add a code comment explaining why its being skipped.

**Important**: Process every issue in the list. Do not
   skip or ignore any issue silently.

At the end, summarize:
- Total number of issues processed
- Number of issues fixed
- Number of issues skipped
```

2. **Autonomous Remediation**: Using its internal agentic capabilities, Cursor Pro processed each issue sequentially. The AI parsed the issue metadata, located the corresponding line in the specified file, generated a fix inline using best practices, and added comments if uncertainty existed (e.g., `// SonarQube false positive S6440: BE.useCoupon is not a React Hook but a backend`

`API method`). Cursor Pro provided diffs between the original and modified code, allowing the user to preview and approve changes.

3. **Manual Review**: Each proposed fix was reviewed manually due to the absence of complete test coverage in the MyFlavoria PWA. Acceptance criteria included preservation of functional behavior, adherence to codebase style, and improvement in maintainability or removal of security flaws. Changes were either accepted most of the times, or reverted, with reasons (e.g., the issue had already been fixed during the previous batch, and now Cursor is adding the comment regarding that).

4. **Post-Fix Validation**: After all issue batches were processed, the updated codebase was rescanned with SonarQube and Snyk to measure the quantitative reduction in issues, detect any newly introduced problems, and identify false positives from the initial scan.

The remediation workflow is illustrated in Figure 4.2, showing the sequence of steps from issue ingestion to validation.

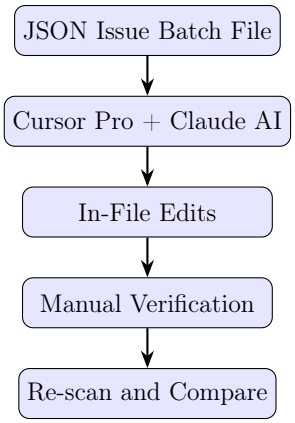

Figure 4.2: AI-driven remediation workflow for processing technical debt issues.

4.4 Evaluation Metrics

This section outlines the quantitative and qualitative metrics used to evaluate the effectiveness of of Cursor Pro, powered by Claude Sonnet 4, in remediating the technical debt issues identified by SonarQube and Snyk. The main goal behind this evaluation was to address the research questions specifically RQ3 (To assess how actionable and comprehensive technical debt insights were from these tools) and RQ4 (To measure the accuracy and effectiveness of AI-driven remediation), and to address the research gap in LLM-Powered Agentic AI technical debt repayment.

4.4.1 Quantitative Metrics

Quantitative metrics were defined to measure the outcomes of the AI-remediation process. The following were used:

- **Issue Resolution Rate**: The percentage of issues (code smells, bugs, vulnerabilities) identified by SonarQube and Snyk that were successfully resolved by Cursor Pro. This was calculated by comparing the number of issues marked as resolved in post-remediation scans to the total number of issues in the initial scans, categorized by issue type.

- **Newly Introduced Issues**: The number of new issues detected in post-remediation SonarQube and Snyk scans, indicating potential errors introduced by Cursor Pro's fixes.

- **Unresolved Issues**: The number of issues that remained unresolved after remediation, calculated as the difference between initial issues and resolved issues.

- **False Positive Rate**: The percentage of issues identified by SonarQube and flagged as false positives by Cursor Pro, based on its context-aware analysis of

the codebase. This was determined by manual review of Cursor's justifications for flagging false positives.

- **Code Quality Improvement**: The improvement in code maintainability, measured using SonarQube's maintainability index and cyclomatic complexity scores before and after remediation.

- **Time Savings**: This indicates the reduction in time required to remediate issues as compared to manual remediation time. Manual remediation time was estimated by the SonarQube to perform the required fixes to repay technical debt issues.

The quantitative metrics such as issues resolution rate, newly introduced issue, unresolved issues, and false positive issues were calculated using data from pre- and post-remediation process of SonarQube and Snyks scans. Code quality improvements were derived from SonarQube's maintainability index and cyclomatic complexity metrics. Time savings were calculated by comparing SonarQube's estimated time with Cursor Pro's processing time recorded by the researcher.

4.4.2 Qualitative Metrics

Qualitative metrics were defined to assess the subjective quality and usability of Cursor Pro's remediation. The following metrics were used:

- **Fix Quality**: The extent to which Cursor Pro's fixes adhered to best practices and preserved functional behavior, evaluated through manual review by researcher. Feedback was recorded which reflects whether fixes were idiomatic and effective.

- **Usability**: The ease of integrating Cursor Pro into the remediation workflow, assessed based on the clarity of diffs, responsiveness to prompts, and overall user experience during manual review.

Qualitative metrics were collected through manual review and false positives flagged by Cursor Pro were validated by comparing it's context-aware deep codebase analysis to SonarQube static checks, with examples documented for analysis.

This experimental design framework allowed for comprehensive analysis in the next chapter, including empirical evidence to RQ3 and RQ4, while also informing strategic recommendations in the Chapter 6.

5 Results and Analysis

This chapter presents the empirical results derived from the evaluation framework established in Chapter 4. The main focus of this chapter is to demonstrate the measurable impact of using Agentic AI system (Cursor Pro) powered by large language model (Claude Sonnet 4) in repaying technical debt to provide insights. The results directly addresses research questions RQ3 (the use of modern tools for technical debt assessment) and RQ4 (the effectiveness of LLM-powered agentic AI in remediation), by providing quantitative insights (e.g issues resolution rate, false positive identifications) and qualitative insights (e.g. fix quality, tool limitations).

The analysis is presented in two main sections:

1. Raw results including false positives
2. Cleaned results after removing false positives

Visualizations are also added to support quantitative analysis and illustrate the impact, and also specific examples such as false positives to enrich the qualitative analysis.

5.1 Dataset Overview

The initial analysis using SonarQube and Snyk CLI identified a total of 163 technical debt issues in the MyFlavoria PWA codebase, which comprised approximately 7,000

lines of code. These issues were comprised of 134 code smells, 25 bugs, and 4 vulnerabilities.

The issues were normalized into a unified JSON schema, as described in Section 4.2, to support consistent formatting and facilitate structured prompting within Cursor Pro. To accommodate Claude Sonnet 4's context window, the issues were split into 17 batches, each containing 10 issues, resulting in files named `issues_part_1.json` to `issues_part_17.json`. This segmentation ensured manageable processing while maintaining semantic coherence where possible. The dataset contained a wide range of severity levels (BLOCKER, CRITICAL, MAJOR, MINOR, INFO) and issue types for evaluating the AI's remediation capabilities in a real-world Typescript-based PWA context.

5.2 Phase 1: Raw Results (Including False Positives)

This section presents the results from processing all 163 issues through Cursor Pro, including false positives identified during manual review. The analysis focuses on remediation outcomes, total issue counts, and distributions by severity and type, providing a baseline for the AI's performance before cleaning the dataset.

5.2.1 Remediation Outcomes

Cursor Pro processed each batch of issues using the prompt template outlined in Section 4.3.2, generating fixes, identifying false positives, and providing diffs for manual review. The outcomes for the 163 issues were as follows:

Table 5.1: Remediation Outcomes (Including False Positives)

Outcome Type	**Count**	**% of Total**
Resolved by AI	135	76.7%
Unresolved issues	28	15.9%
New Issues Introduced	13	7.4%

The Cursor Pro autonomously resolved 135 issues (76.7%), demonstrating significant capability in addressing technical debt without human intervention. Additionally, 23 issues were identified as false positives by Cursor Pro which left unresolved, primarily due to its context-aware analysis outperforming SonarQube's static checks. 13 new issues were introduced, identified during post-remediation scans, indicating issues in the code fixes that AI provided. Figure 5.1 visualizes the distribution of these outcomes.

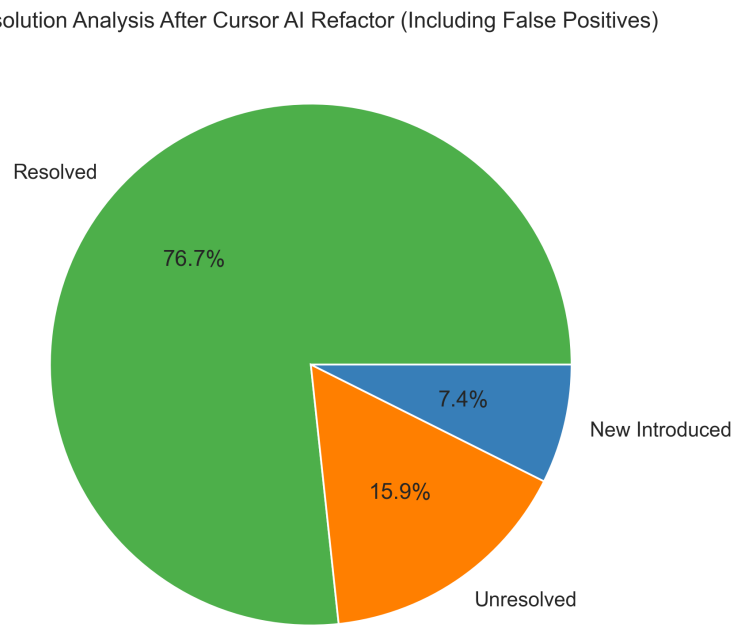

Figure 5.1: Remediation Outcome Distribution (Including False Positives). The pie chart shows the proportion of issues resolved by AI, issues which left unresolved and newly introduced.

5.2.1.1 Analysis of Newly Introduced Issues

While Cursor Pro effectively resolved most of the technical debt issues, the remediation process introduced 13 new issues which were primarily related to code smell rather than functional correctness. These new issues highlight trade-offs when using AI-based tools for technical debt repayment. Some of the newly introduced issues include:

- **Improper Operator Use**: The AI used the logical OR operator (||) instead of the nullish coalescing operator (??) in multiple files when fixing the issues. Although both serve similar purpose but ?? is safer when dealing with null or undefined to avoid unintended fallbacks for falsy values like 0 or empty strings ("").

- **Increased Cognitive Complexity**: In some fixes, the AI introduced cognitive complexity which exceeded the recommended threshold by SonarQube, and made the code harder to read and reason about. Similarly, in one fix, the nesting depth of a function surpassed the acceptable limit, which lead to decreased clarity and poor readability.

Though many existing issues were resolved by AI, but some new issues, often related to stylistic preference or structural code quality, may introduce. This highlights the need for post-AI-review audits by developers to ensure that the fixes are aligned with project standards and maintainability goals.

5.2.1.2 Analysis of Issues Left Unresolved

Despite the effectiveness of AI, some issues remained unresolved in the codebase. These issues typically required human context and architectural foresight since current LLM-based tools may lack full domain understanding and decision making capabilities. The following issues specifically remained unaddressed:

- **TODO and FIXME comments**: A TODO comment in a file indicated an incomplete feature or logic, and AI did not implement that feature, since it requires domain-specific decisions. Similarly, a FIXME comment at one place highlighted a limitation in multilingual support logic which mentioned that the implementation may not scale for additional languages. The AI avoided modifying this logic to prevent breaking changes and added the reason for that. Figure 5.2 and Figure 5.3 illustrate the TODO and FIXME issues examples respectively.

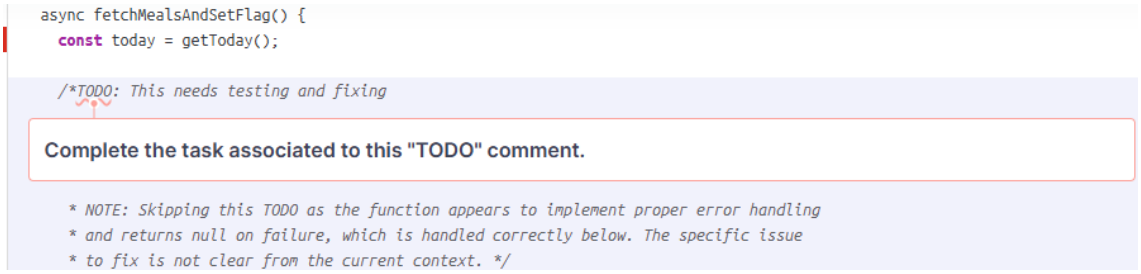

Figure 5.2: TODO issue example

Figure 5.3: FIXME issue example

5.2.1.3 False Positive Examples

Cursor Pro identified 23 false positives, leveraging its context-aware analysis. For example, SonarQube flagged a property to be initialized with 'readonly' as a code smell, but Cursor Pro marked this as false positive due to it's context aware analysis and justified it as intentional because it was initialized lazily in the file. Figure 5.4 illustrates this example.

```
class BackendFactory {
  // Note: Cannot make this readonly as it's lazily initialized in getBackend()
  static backend: AbstractBackend | null = null;

  Make this public static property readonly.

  static getBackend(): AbstractBackend {
    BackendFactory.backend ??= debug
      ? new DebugBackend()
      : new MyFlavoriaBackend();
    return BackendFactory.backend;
  }
}
```

Figure 5.4: False Positive 1

Similarly, SonarQube identified the string literal starting with 'use' to be the React Hook and flagged that as Bug, but Cursor Pro marked that a false positive because that was a Backend API which was being called. Figure 5.5 illustrates this example.

```
    if (debugCoupons) {
      console.log(currentlySelected);
      setCurrentlyActive(currentlySelected);
    } else {
      // SonarQube false positive S6440: BE.useCoupon is not a React Hook but a backend API method
      BE.useCoupon(currentlySelected).then(() => {

React Hook "BE.useCoupon" is called in function "activateCurrentCoupon" that is neither a React function component nor a custom React Hook function. React component names must start with an uppercase letter. React Hook names must start with the word "use".

        setCurrentlyActive(currentlySelected);
      });
    }
```

Figure 5.5: False Positive 2

Another example where SonarQube identified the incorrect use of useState hooks multiple time in different files which it indicated that it does not contain value+setter pair, but Cursor Pro marked those as false positive because those were actually properly initialized and contained proper value+setter pair. Figure 5.6 illustrates one of the examples from those issues.

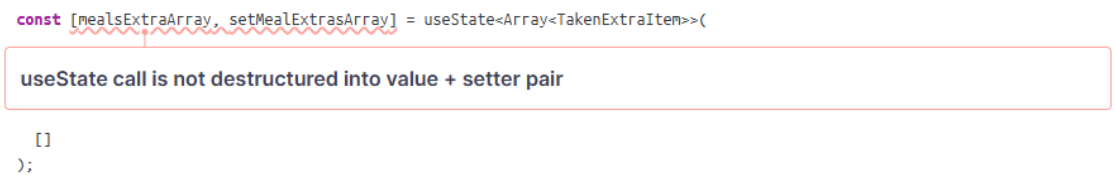

Figure 5.6: False Positive 3

These issues which were flagged false positive by Cursor were validated manually, which confirmed their false status. The next section includes the analysis after removing those false positive issues identified by Cursor for cleaned analysis.

5.2.2 Total Issue Comparison: Before vs After AI Fix

The initial 163 issues were reduced significantly after AI remediation, though some of the issues were left unresolved which were detected as false positives and 13 new issues slightly reduced the net accuracy. Post-remediation scans with SonarQube and Snyk reported 41 remaining issues, including the 13 new issues. Figure 5.7 illustrates this comparison.

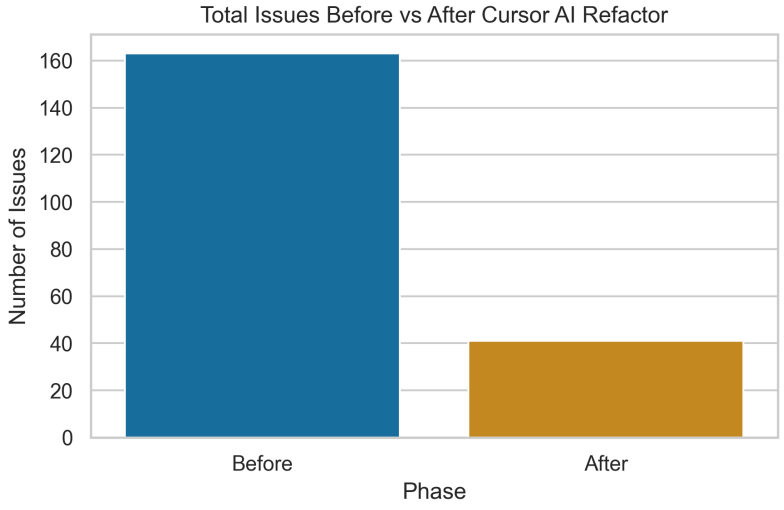

Figure 5.7: Total Issue Count Before and After AI Fixes. The bar chart compares the initial 163 issues to the 41 remaining issues post-remediation.

5.2.3 Issue Type Distribution (Before vs After)

The issue types included code smells, security vulnerabilities, and bugs. Table 5.2 summarizes the counts by type.

Table 5.2: Issue Type Distribution Before and After Remediation

Phase	Code Smell	Security	Bug
Before	134	4	25
After	37	2	2

Significant reductions were observed in code smells (from 134 to 37) and bugs (from 25 to 2), with less improvements in security vulnerabilites. Figure 5.8 presents this breakdown.

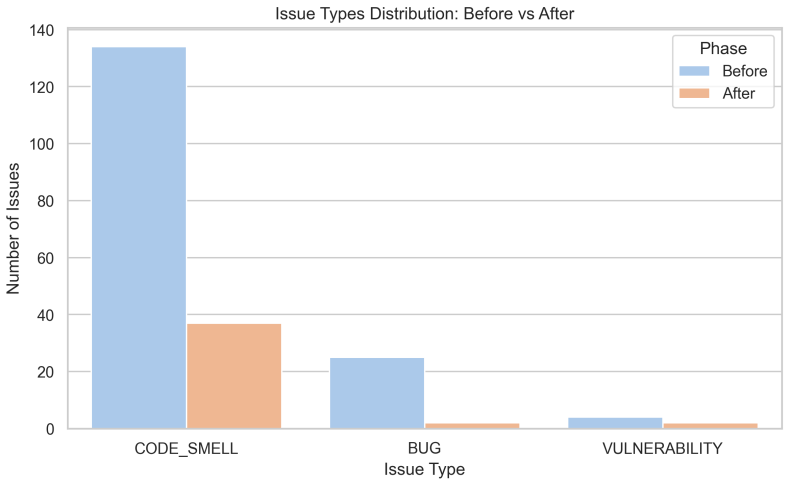

Figure 5.8: Issue Type Breakdown – Before vs After Remediation. The grouped bar chart highlights reductions in code smells and bugs.

5.3 Phase 2: Cleaned Analysis (Excluding False Positives)

After manual validation, 23 issues were marked as false positive which were classified as false positives by Cursor, to analyze the AI's performance on the cleaned set of issues, providing a more accurate assessment of its remediation capabilities.

5.3.1 Issue Totals After Cleaning

The cleaned dataset showed a clearer picture of the AI's effectiveness. Figure 5.9 illustrates the issues count of total before Cursor AI fix and after Cursor AI fix (excluding false positives by manual review of those issues flagged by Cursor AI as false positive). After Cursor AI fix, there were 41 issues remaining, out of which 23 were flagged as false positive, which reduced the actual issues to 18.

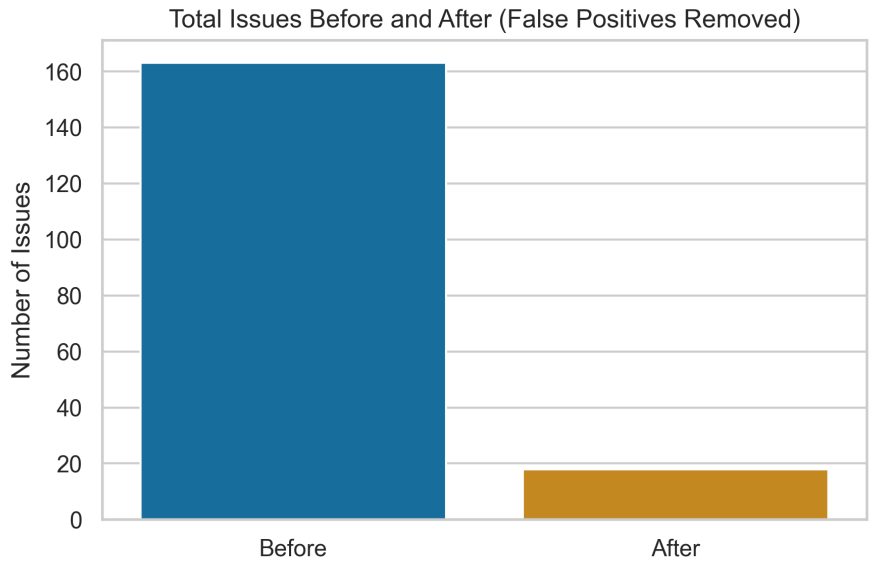

Figure 5.9: Total Issues – Before vs After (Cleaned).

5.3.2 Resolution Outcome (Cleaned)

The cleaned remediation outcomes, excluding false positives, were recalculated as follows:

Table 5.3: Remediation Outcomes (Excluding False Positives)

Outcome Type	**Count**	**% of Total**
Resolved by AI	156	(89.7%)
Unresolved Issues	7	(4.0%)
New Issues Introduced	11	(6.3%)

Figure 5.10 visualizes the updated outcomes.

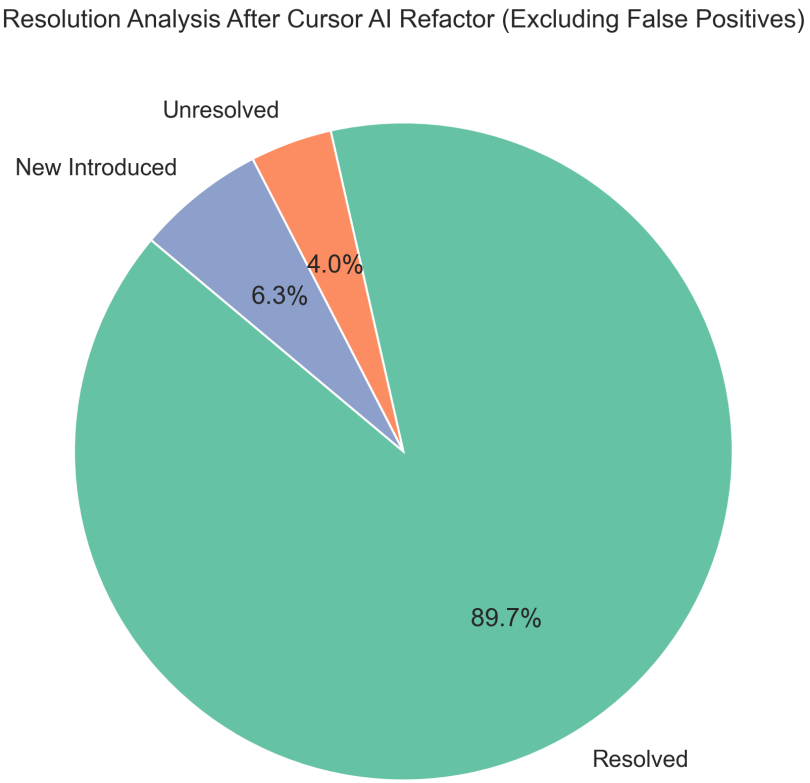

Figure 5.10: Cleaned Remediation Outcomes (Excluding False Positives)

The cleaned resolution outcome represents a more accurate picture of AI's effectiveness after accounting for false positives. Over 89% of issues were successfully resolved by the agentic AI, reflecting a cleaned resolution success rate. The number is significantly higher than the raw success rate (76.7%) reported in the previous section, emphasizing the importance of filtering out irrelevant or invalid findings when evaluating AI remediation effectiveness.

5.4 Code Quality Improvement and Time Savings

This section shows the improvements in code quality and maintainability scores after automated fixes using Cursor AI, as measured by SonarQube. The metrics are calculated based on before and after measurements of applying the AI fixes. The aim was to measure the effectiveness of large language model (LLM)-powered automation in technical debt reduction and improving developer efficiency.

Significant reductions were observed in key technical debt indicators calculated from SonarQube's before and after analysis. Table 5.4 summarizes the overall assessment counts.

Metric	**Before**	**After**	**Improvement**
Bugs	25	2	-92%
Code Smells	134	37	-72%
Cognitive Complexity	546	486	-11%
Cyclomatic Complexity	830	822	Slight
Vulnerabilities	1	0	Fully Resolved
Security Rating	E (5.0)	A (1.0)	Improved
Maintainability Index (SQALE Index)	1323	209	-84%

Table 5.4: SonarQube's Software Quality Metrics Before and After Improvements

These results indicate enhanced maintainability and reduced technical debt, specifically a huge drop in the SQALE index, which is the estimated measure of effort in minutes by SonarQube needed to fix issues. The estimated time was around 22 hours (1323 minutes), but it took less than 2 hours to fix those issues through Cursor AI. The automated remediation performed by Cursor AI significantly reduced the developer effort required for bug fixing and code simplification. The change in the security rating from worst (E) to best (A) also indicates successful mitigation of security vulnerability.

While there was 11% reduction in cognitive complexity, but the numbers are still high which indicates that significant complexity still exists in the codebase. The minimal change in cyclomatic complexity also suggests that the control flow logic and branching structures were largely unaffected after AI remediation. This reflects limited effectiveness of AI in simplifying complex logic and improving structural design.

6 Discussion

This chapter reflects on the findings related to the research questions introduced in this thesis and results presented in Chapter 5, and connects them to give broader context of software maintainability and AI-assisted development. The discussion highlights the strengths and limitations of the approach used, reflects on the role of technical debt remediation tools and strategies in practice in terms of ongoing development in AI-powered software engineering specifically regarding maintainability, developer productivity and technical debt repayment, and explores opportunities for future research.

6.1 Overview of Results

The primary goal of this study was to evaluate how modern tools, particularly agentic AI systems, can help in technical debt remediation process within real-world software systems.

The results show that agentic AI system successfully resolved a major portion of technical debt issues especially code smells and bugs. It was also capable of detecting and reasoning about false positives by adding justified comments when fixes were unnecessary or inappropriate. However, the need for manual validation, the AI's limited effectiveness in resolving cyclomatic complexity, and understanding and fixing issues requiring architectural decisions and more domain understanding highlight the current limitations. The discussion below addresses each research

question in light of these findings.

6.2 Addressing the Research Questions

6.2.1 RQ1: Development Practices Leading to Technical Debt

The literature review along with the analysis of MyFlavoria system under case study highlighted several development practices contributing to technical debt accumulation.

The legacy mobile application used many dependencies which were deprecated and package versions were incompatible with latest versions of libraries and frameworks, which led to reduced maintainability and blocked upgrades. This aligns with research by Kula et al. [28], who reported that over 80% of Java/Maven projects have outdated dependencies due to extra work required to update them. Automated tools like Dependabot or Renovate were not integrated in CI/CD workflow. Due to this, outdated dependencies were not flagged and updated regularly, which created compatibility risks and a complete redevelopment of the app. Studies by Alfadel et al. [58] and Runzhi he et al. [68] also confirm that automated dependency management significantly reduces technical lag and vulnerabilities.

Inadequate inline comments and no test coverage made it difficult to verify and understand existing logic especially for AI agent in reducing complexity in the code. This aligns with Kruchten's classification of invisible debt, especially in the code-level granularity [13]. The system also contained signs of quick delivery such as TODO/FIXME comments, minimal error handling and inconsistent naming conventions. These patterns align with the Fowler's quadrant model [12], especially the reckless-deliberate debt caused by knowingly taking shortcuts in fully aware situation of long-term consequences.

The application contained number of code smells and bugs which violated good

coding standards and introduced code debt. Though the functionality of the app was not affected with the violation of coding standards, but it makes harder to make even a small update in the future. Importantly, these development practices can be effectively mapped to McConnell's debt classification (Figure 2.1), where these identified issues fall under Unintentional Debt. These issues are not introduced maliciously or strategically, but rather emerge from time pressure, limited team capacity or inexperience.

All of the identified issues align with the broader findings from literature and reflect multiple types of technical debt in the system such as code debt, architectural debt and test debt, as previously discussed in Section 2.1.2. The results support global findings such as those from the InsighTD survey [18], which identified rework, low maintainability, need of refactoring and bad code as the effects of TD.

6.2.2 RQ2: Strategies to Mitigate Long-Term Maintainability Issues

While this thesis primarily focused on the remediation of existing identified technical debt, the findings from the MyFlavoria case study when combined with insights from the literature suggest several actionable strategies that can help prevent or reduce long-term accumulation of maintainability issues in real-world software systems.

First and foremost, the integration of static code analysis tools such as SonarQube and security scanners like Snyk into early development phases is critical. These tools not only detect code smells, bugs, and vulnerabilities, but also provide continuous feedback on maintainability metrics like code complexity, duplicates, and security risks. Integrating them into CI/CD pipelines ensures that developers are made aware of technical debt as it is introduced, which allows for earlier and less costly remediation. This approach aligns with best practices recommended in prior studies [6], [9], which emphasize that real-time monitoring of technical debt using

automated tools is an important step in proactive technical debt management.

Secondly, maintainability can be significantly improved by automating dependency management. Tools like Dependabot and Renovate can automate this process by regularly scanning for outdated libraries and automatically generating pull requests for safe upgrades. This strategy has been validated in studies such as [58] and [68], which found that automation significantly reduces both the volume and risk associated with dependency-related debt.

Perhaps the most forward-looking strategy involves the use of agentic AI tools, such as Cursor Pro powered by Claude Sonnet 4, which were used in this study to remediate code-level technical debt. These systems are not only capable of detecting issues but also autonomously applying fixes, which reduces the manual burden on developers and accelerates debt repayment. In this study, over 89% of code-level issues were successfully resolved by the agent, demonstrating that such AI can meaningfully contribute to long-term maintainability. This finding aligns with recent advancements in automatic program repair and LLM-driven remediation [3], [4], [5], which show that AI models are increasingly capable of understanding context, identifying anti-patterns, generating corrective code changes and resolving quality issues.

To summarize, combining preventive monitoring via integrating scanners into CI/CD, automated dependency management, and autonomous remediation using agentic AI offers a multi-layered strategy for mitigating long-term maintainability debt. When used together, these techniques provide a proactive approach that not only prevents the introduction of technical debt but also ensures that it is addressed swiftly and systematically when it does occur.

6.2.3 RQ3: Using Modern Tools to Assess Maintainability, Reliability, and Security

The experimental methodology demonstrated that tools like SonarQube and Snyk offer detailed actionable insights in identifying maintainability and security related issues.

Since SonarQube uses rule-based static code analysis, it provides comprehensive analysis of bugs, code smells, vulnerabilities, cyclomatic and cognitive complexity metrics. It also has capabilities of integration within the modern CI/CD workflows. It also provides comparative scans history to analyze before-and-after scans results which served as a base to evaluate the effectiveness of the proposed methodology in this study. Though SonarQube identified only 1 security vulnerability in the codebase, Snyk detected 3 vulnerabilities, which highlights the importance of using tools specifically tailored for code security testing.

Their outputs were structured into unified JSON format for chunk-wise processing by Cursor Pro, which demonstrate a novel pipeline to incorporate static code analyzers and security code scanners within agentic AI systems, which is not widely explored in previous literature as mentioned in Section 2.3.4.

However, these tools do have limitations as well. Due to the lack of contextual awareness and deep codebase reasoning, SonarQube detected many issues which were classified as false positive by AI during remediation process. Those issues were confirmed as false positives during manual review process. This highlights the importance of human review in remediation process to confirm these issues so that the app's required functionality is not broken.

These tools acted as code reviewers and foundational components for AI-driven remediation pipeline providing structured issues data that guided the LLM's actions.

6.2.4 RQ4: Effectiveness of LLM-powered AI in remediating technical debt

The use of Cursor Pro along with Claude Sonnet 4 has shown promising results in remediating technical debt.

Out of 163 identified issues, almost 89% issues were resolved by the AI. Due to context awareness and deep codebase reasoning capabilities, Cursor marked almost 23 issue as false positive which were confirmed after manual review. The comments were added to justify why the issue was flagged as false positive and/or why it is left unresolved. Cursor was able to handle chunk of issues at same time by navigating to correct files and lines to make changes. This experimental process significantly reduced time effort required to remediate the identified issues as compared to manual estimated work time. The quality of the fixes was confirmed by the developer involved in maintaining the application under case study, but he pointed out that one fix has changed the logic which caused new bugs in the file. This highlights the importance of manual human review in the loop even after fixes are generated by the AI.

These results validate the core hypothesis developed in this thesis that LLM-based agentic AI systems can significantly reduce technical debt repayment. Similar results were seen in a study of Codex based bug fixing experiments [3], [4], [5]. Microsoft's CORE [45] tool reported success rate of up to 76% in Java projects but only targeted bug fixing. But, this study also addressed vulnerabilities along with comprehensive software quality metrics using an agentic-AI system.

However, there are some limitations which were noticed during this experiment. The AI did not resolve issues related to cognitive and cyclomatic complexities, as well as the issues like TODO/FIXME, which required more context and domain-specific decisions. Due to shorter context window, issues were divided into smaller chunks containing 10 issues per file to manage the context window and easily interpret

the results of the fixes. Though based on the results, agentic AI offers value in remediating issues but human oversight is still required for context-sensitive issues. So, the strategy of AI-assisted remediation with human validation provides a path forward to incorporate this into software development workflow.

6.3 Study Limitations

While this thesis contributes novel insights about AI-powered technical debt remediation and maintainability strategies, there are some limitations to this study which should be acknowledged. Due to time constraints and scope of this thesis, the findings are based solely on single case study application MyFlavoria PWA, which is a mid-sized React based application. The results may not generalize to other architectures, different tech stacks or different programming languages which should be explored and evaluated to get overall generalized insights of this experimental approach.

The technical debt remediation process was performed only using Cursor Pro (which is a state-of-the-art agentic IDE) with Claude Sonnet 4 (advanced and latest LLM). So, the results obtained in this study are specific to this selected tool and LLM. Other LLMs such as GPT-4, Gemini etc. may have different performance capabilities, so the findings of this study do not apply as universal effectiveness across all AI tools and LLMs.

The primary evaluation of technical debt assessment was measured using SonarQube and Snyk. Though these tools are widely used but other tools listed in the previous chapters may have different capabilities in specific domain or language. All of the listed tools are not tested for identifying technical debt issues, so the results obtained are specifically tied to the selected tools. Although Snyk was used to identify vulnerabilities in the code but the depth of the security assessment was limited since only 4 issues were identified in the application under case study. So evaluation

of these tools on large-sized enterprise level applications is still not explored. Also, the process was not end-to-end automated since the remediation workflow required human validation in the process of the fixes. The reliance on developer judgement also affects reproducibility across teams with varying experience levels.

Although this thesis did not experimentally analyze the dependency related debt, but the literature presented it's strong relevance in the debt accumulation. Tools were presented in the Section 3.3 to support dependency management tasks but they were not explored in this study. A future direction could involve combining static/security code analyzers, dependency updaters along with agentic AI fixers into an integrated maintainability workflow.

6.4 Future Directions

This thesis has demonstrated the potential of using AI-driven approaches in technical debt remediation, but there are still many opportunities to expand and improve this work. First, the proposed methodology could be extended and validated on larger, enterprise-level codebases and across a broader range of programming languages (e.g., Python, Java). The current study primarily focused on mid-sized React application. So, applying the approach to more diverse and complex real-world software systems would help assess its scalability and generalizablility in real-world industrial environments.

Second, as the landscape of the large language models (LLMs) is evolving rapidly, a comparative evaluation of both closed-source and open-source models would be highly valuable. Closed-source models such as OpenAI's GPT-4 and Google's Gemini are known for their high performance but often lack transparency and flexibility. In contrast, open-source alternatives such as Mistral, LLaVA, Meta's LLaMA and Google's Gemma offer greater customizability and local deployment capabilities, which can be advantageous in enterprise or privacy-sensitive environments. Compar-

ing these models in technical debt remediation process could provide insights about performance, cost, and limitations which can serve as a base for model selection suitable for specific requirements of different software development environments.

Third, integrating AI-based remediation tools directly into continuous integration and deployment (CI/CD) pipelines represents a critical step toward practical adoption. Integrating the remediation process in CI/CD and maintainability workflow along with automated test validation, could enable real-time debt detection and correction while ensuring the functional correctness of proposed code changes.

In addition, the use of alternative AI-assisted development tools beyond Cursor could also be explored. While Cursor has proven effective in this study, tools such as WindSurf, Void and Cline (both are open-source) offer distinct user experiences and integration capabilities. Evaluating these tools in the context of technical debt remediation could provide a broader understanding of how developer-facing AI solutions support remediation in practice.

Finally, future work could explore the remediation capabilities beyond code-level issues to include architectural and dependency-related technical debt. These types of debt often have long-term implications for system maintainability, performance and scalability. So, AI models may be beneficial in addressing these areas as they continue to evolve.

7 Conclusion

The primary aim of this thesis was to investigate how technical debt, particularly related to maintainability and security, can be identified, analyzed, and remediated using modern tooling and AI-driven approaches. Through an in-depth case study analysis, this research presented a diagnostic-proactive approach to managing technical debt by combining traditional tools like SonarQube and Snyk with advanced agentic AI powered by large language models (LLMs), such as Claude Sonnet 4 integrated into the Cursor development environment. To provide a clear summary of the study's contributions, the following paragraphs revisit the research questions and highlights the key insights gained.

Research Question 1 (RQ1): *What development practices contribute to the accumulation of technical debt in software systems?* The findings demonstrate that technical debt in real-world systems is often not the result of poor coding standards alone but emerges from a confluence of bad architectural decisions, neglected outdated dependencies, limited documentation, no test coverage, and short-term development priorities. These observations are consistent with earlier classifications of technical debt introduced in Chapter 2, highlighting both unintentional and short-term deliberate forms of debt. Recognizing these root causes enables more effective planning and prevention strategies.

Research Question 2 (RQ2): *What strategies can be employed to help reduce long-term maintainability issues in real-world systems?* This study identified multiple

strategies for mitigating long-term maintainability issues, including early integration of static and security analysis tools in CI/CD workflows, automated dependency updates, and the use of AI-assisted remediation agents. These strategies collectively aim to reduce the manual burden of maintaining code quality over time and reducing technical debt.

Research Question 3 (RQ3): *How available modern tools can be utilized for technical debt assessment to provide analysis about maintainability, reliability and security of a software?* Modern tools such as SonarQube (static analysis) and Snyk (security scanning) were shown to be valuable in diagnosing debt, specifically code-level debt by providing detailed insights into various maintainability metrics including cyclomatic complexity, code smells, bugs, and security vulnerabilities. Importantly, these tools were not only useful for highlighting issues but also served as a bridge to feed meaningful input into AI-based agents for remediation process, as they provide detailed issue descriptions, including severity classification.

Research Question 4 (RQ4): *How effective is an LLM-powered agentic AI in remediating technical debt issues identified by static code analysis and security scanning tools in software systems, in terms of resolution rate and accuracy?* One of the most significant contributions of this thesis lies in providing a methodology for practical demonstration of integrating agentic AI into a remediation workflow. Claude Sonnet 4 proved capable of resolving a high percentage of identified issues, particularly those involving code smells, bugs, and low-complexity design flaws. The AI's capability to distinguish between the true issues and false positives highlights the growing sophistication of large language models in software maintenance tasks. However, limitations were also observed. The AI agent performed well with low to medium complexity issues but struggled with issues involving architectural decisions, complex logic and domain-specific knowledge. Furthermore, while the agentic AI was capable of flagging false positives, it required manual human oversight to

validate some decisions and outputs. This suggests that while agentic AI is powerful, a hybrid model that includes human validation remains necessary for safe and effective adoption in production environments.

From a broader perspective, this research contributes to the evolving field of AI-assisted software maintenance by proposing and evaluating a novel end-to-end remediation workflow that integrates static analysis, security scanning, structured data transformation, and LLM-powered remediation in a real-world case study.

In conclusion, this research demonstrates that technical debt is a manageable challenge when addressed through a combination of developer awareness, modern tooling, and AI-driven support. Large language models, particularly when used within agentic frameworks and integrated with static and security analysis tools, can meaningfully enhance software maintainability by streamlining remediation efforts and reducing developer burden. While AI is not a complete solution, its ability to automate routine refactoring and issue triage enables developers to focus on higher-level design decisions. Overall, the findings highlight the potential of AI to support more maintainable and secure software systems.

The journey toward fully autonomous maintainability is ongoing, but this work provides a practical and forward-thinking step in that direction. Future advancements in agentic AI and context reasoning may eventually allow these systems to handle more complex forms of technical debt.

References

[1] T. Besker, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work", *Journal of Systems and Software*, vol. 156, pp. 41–61, 2019.

[2] Accenture, "Build your tech and manage your debt", 2024. [Online]. Available: https://www.accenture.com/content/dam/accenture/final/accenture-com/document-3/Accenture-Build-Your-Tech-and-Manage-Your-Debt-2024.pdf.

[3] J. A. Prenner and R. Robbes, *Automatic program repair with openai's codex: Evaluating quixbugs*, 2021. arXiv: 2111.03922 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2111.03922.

[4] M. Lajko, V. Csuvik, T. Gyimothy, and L. Vidacs, "Automated program repair with the gpt family, including gpt-2, gpt-3 and codex", ser. APR '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 34–41, ISBN: 9798400705779. DOI: 10.1145/3643788.3648021. [Online]. Available: https://doi.org/10.1145/3643788.3648021.

[5] D. Sobania, M. Briesch, C. Hanna, and J. Petke, *An analysis of the automatic bug fixing performance of chatgpt*, 2023. arXiv: 2301.08653 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2301.08653.

[6] J. P. Biazotto, D. Feitosa, P. Avgeriou, and E. Y. Nakagawa, *Automating technical debt management: Insights from practitioner discussions in stack exchange*, 2025. arXiv: 2502.03153 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2502.03153.

[7] W. Cunningham, "The wycash portfolio management system", *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, Dec. 1992, ISSN: 1055-6400. DOI: 10.1145/157710.157715. [Online]. Available: https://doi.org/10.1145/157710.157715.

[8] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)", *Dagstuhl Reports*, vol. 6, Jan. 2016. DOI: 10.4230/DagRep.6.4.110.

[9] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management", *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.12.027. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002854.

[10] N. Brown et al., "Managing technical debt in software-reliant systems", Nov. 2010, pp. 47–52. DOI: 10.1145/1882362.1882373.

[11] S. McConnell, *Managing technical debt: A white paper by construx software*, Accessed: 2025-01-27, 2019. [Online]. Available: https://www.construx.com/wp-content/uploads/2019/02/CxWhitePaper_TechnicalDebt.pdf.

[12] M. Fowler, *Technical debt quadrant*, Accessed: 2025-02-03, 2009. [Online]. Available: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

[13] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice", *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012. DOI: 10.1109/MS.2012.167.

[14] P. Quezada Sarmiento, D. Guaman, L. R. Barba Guamán, L. Enciso, and P. Cabrera, "Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis", Jul. 2017.

[15] J. Yeboah and S. Popoola, *Efficacy of static analysis tools for software defect detection on open-source projects*, 2024. arXiv: 2405.12333 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2405.12333.

[16] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt", *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.12.052. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213000022.

[17] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study", *Information and Software Technology*, vol. 70, pp. 100–121, 2016, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.10.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001743.

[18] R. Ramač et al., "Prevalence, common causes and effects of technical debt: Results from a family of surveys with the it industry", *Journal of Systems and Software*, vol. 184, p. 111 114, Feb. 2022, ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111114. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2021.111114.

[19] N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman, "The most common causes and effects of technical debt: First results from a global family of industrial surveys", in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM

'18, New York, NY, USA: Association for Computing Machinery, 2018, ISBN: 9781450358231. DOI: 10.1145/3239235.3268917. [Online]. Available: https://doi.org/10.1145/3239235.3268917.

[20] R. R. de Almeida, C. Treude, and U. Kulesza, *What's behind tight deadlines? business causes of technical debt*, 2023. arXiv: 2104.09330 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2104.09330.

[21] V. Lenarduzzi, T. Orava, N. Saarimäki, K. Systä, and D. Taibi, *An empirical study on technical debt in a finnish sme*, 2019. arXiv: 1908.01502 [cs.SE]. [Online]. Available: https://arxiv.org/abs/1908.01502.

[22] R. Banker, Y. Liang, and N. Ramasubbu, "Technical debt and firm performance", *Management Science*, vol. 67, no. 5, pp. 3174–3194, 2020. DOI: 10.1287/mnsc.2019.3542.

[23] G. Suryanarayana, S. Ganesh, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Nov. 2014, pp. 1–237, ISBN: 978-0128013977.

[24] M. Dawson, D. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)", *Journal of Information Systems Technology and Planning*, vol. 3, pp. 49–53, Jan. 2010.

[25] E. Sutoyo, P. Avgeriou, and A. Capiluppi, "Tracing the lifecycle of architecture technical debt in software systems: A dependency approach", *arXiv preprint arXiv:2501.15387*, 2025.

[26] H. Nilsson and L. Petersson, "How to manage technical debt in a lean startup", M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden, 2013. [Online]. Available: https://publications.lib.chalmers.se/records/fulltext/216789/216789.pdf.

[27] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, Inc., 2020, p. 602, ISBN: 1492082767, 9781492082767.

[28] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration", *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, May 2017, ISSN: 1573-7616. DOI: 10.1007/s10664-017-9521-5. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9521-5.

[29] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems", *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, Jul. 2021, ISSN: 1049-331X. DOI: 10.1145/3447245. [Online]. Available: https://doi.org/10.1145/3447245.

[30] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications", in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1513–1531, ISBN: 9781450370899. DOI: 10.1145/3372297.3417232. [Online]. Available: https://doi.org/10.1145/3372297.3417232.

[31] J. P. Biazotto, D. Feitosa, P. Avgeriou, and E. Y. Nakagawa, "Technical debt management automation: State of the art and future perspectives", *Information and Software Technology*, vol. 167, p. 107 375, Mar. 2024, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2023.107375. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2023.107375.

[32] S. B. Pandi, S. A. Binta, and S. Kaushal, *Artificial intelligence for technical debt management in software development*, 2023. arXiv: 2306.10194 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2306.10194.

[33] SonarSource, *Sonarqube*, Accessed: 2025-05-07, 2025. [Online]. Available: https://www.sonarsource.com/products/sonarqube/.

[34] Y. Almeida et al., "Aicodereview: Advancing code quality with ai-enhanced reviews", *SoftwareX*, vol. 26, p. 101 677, 2024, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2024.101677. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711024000487.

[35] Z. Li et al., "Automating code review activities by large-scale pre-training", in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1035–1047, ISBN: 9781450394130. DOI: 10.1145/3540250.3549081. [Online]. Available: https://doi.org/10.1145/3540250.3549081.

[36] Y. Li, M. Soliman, P. Avgeriou, and M. Van Ittersum, " DebtViz: A Tool for Identifying, Measuring, Visualizing, and Monitoring Self-Admitted Technical Debt ", in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2023, pp. 558–562. DOI: 10.1109/ICSME58846.2023.00072. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSME58846.2023.00072.

[37] V. V. Mangave, "Automated Code Review Tool", *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT*, vol. 08, no. 05, pp. 1–5, May 2024. DOI: 10.55041/ijsrem33572.

[38] E. L. Melin and N. U. Eisty, *Exploring the advances in using machine learning to identify technical debt and self-admitted technical debt*, 2024. arXiv: 2409.04662 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.04662.

[39] M. Chen et al., *Evaluating large language models trained on code*, 2021. arXiv: 2107.03374 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2107.03374.

[40] X. Li, *A review of prominent paradigms for llm-based agents: Tool use (including rag), planning, and feedback learning*, 2024. arXiv: 2406.05804 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2406.05804.

[41] C. Sypherd and V. Belle, *Practical considerations for agentic llm systems*, 2024. arXiv: 2412.04093 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2412.04093.

[42] Windsurf, *Windsurf editor*, https://windsurf.com/editor, Accessed: 2025-05-23, 2025.

[43] Anysphere Inc., *Cursor: The ai code editor*, https://www.cursor.com/, Accessed: 2025-05-23, 2025.

[44] E. Sutoyo and A. Capiluppi, *Self-admitted technical debt detection approaches: A decade systematic review*, 2024. arXiv: 2312.15020 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2312.15020.

[45] N. Wadhwa et al., "Core: Resolving code quality issues using llms", in *FSE*, Jul. 2024. [Online]. Available: https://www.microsoft.com/en-us/research/publication/core-resolving-code-quality-issues-using-llms/.

[46] Semaphore. "How to use ai to reduce technical debt". Accessed: 2025-05-25. [Online]. Available: https://semaphoreci.medium.com/how-to-use-ai-to-reduce-technical-debt-02f7786884ac.

[47] J. Liu et al., *Large language model-based agents for software engineering: A survey*, 2024. arXiv: 2409.02977 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.02977.

[48] P. C. Avgeriou et al., "An overview and comparison of technical debt measurement tools", *IEEE Software*, vol. 38, no. 3, pp. 61–71, 2021. DOI: 10.1109/MS.2020.3024958.

[49] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners", *Information and Software Technology*, vol. 102, pp. 117–145, 2018, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.05.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918300946.

[50] P. Avgeriou et al., "Technical debt management: The road ahead for successful software delivery", in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, IEEE, May 2023, pp. 15–30. DOI: 10.1109/icse-fose59343.2023.00007. [Online]. Available: http://dx.doi.org/10.1109/ICSE-FoSE59343.2023.00007.

[51] CAST, *Cast imaging*, Accessed: 2025-04-14, 2025. [Online]. Available: https://www.castsoftware.com/imaging.

[52] NDepend, *Ndepend*, Accessed: 2025-04-14, 2025. [Online]. Available: https://www.ndepend.com/.

[53] Vector Informatik, *Squore: Augmented analytics for efficient project monitoring*, Accessed: 2025-04-14, 2025. [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/squore/.

[54] Codacy, *Codacy: Code quality and security for developers*, Accessed: 2025-04-14, 2025. [Online]. Available: https://www.codacy.com/.

[55] Designite, *Designite: Software design quality assessment tool*, Accessed: 2025-04-14, 2025. [Online]. Available: https://www.designite-tools.com/.

[56] SAP, *Sap code inspector*, Accessed: 2025-04-14, 2025. [Online]. Available: https://help.sap.com/doc/saphelp_nw75/7.5.5/en-US/49/205531d0fc14cfe10000000a421frameset.htm.

[57] Symfony, *Symfonyinsight: Quality assurance tool for symfony projects*, Accessed: 2025-04-14, 2025. [Online]. Available: https://insight.symfony.com/.

[58] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests", in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 254–265. DOI: 10.1109/MSR52588.2021.00037.

[59] *Dependabot*, https://github.com/dependabot, Accessed: 2025-07-07.

[60] *Renovate bot*, https://github.com/renovatebot, Accessed: 2025-07-07.

[61] *Pyup*, https://pyup.io, Accessed: 2025-07-07.

[62] *Snyk bot*, https://github.com/snyk-bot, Accessed: 2025-07-07.

[63] M. Wyrich, R. Ghit, T. Haller, and C. Müller, "Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests", in *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, 2021, pp. 6–10. DOI: 10.1109/BotSE52550.2021.00009.

[64] *Safety cli cybersecurity research*, https://www.getsafety.com/cli, Accessed: 2025-07-07.

[65] L. Erlenhov, F. G. de Oliveira Neto, and P. Leitner, "Dependency management bots in open-source systems—prevalence and adoption", *PeerJ Computer Science*, vol. 8, e849, 2022. DOI: 10.7717/peerj-cs.849. [Online]. Available: https://doi.org/10.7717/peerj-cs.849.

[66] G. Baker, *Acquired by github!*, Accessed: 2025-04-14, Jun. 2019. [Online]. Available: https://www.indiehackers.com/product/dependabot/acquired-by-github--LgT7DN1rGEZM2O4srhF.

[67] M. McDonald, *Goodbye dependabot preview, hello dependabot!*, Accessed: 2025-04-14, Apr. 2021. [Online]. Available: https://github.blog/news-insights/product-news/goodbye-dependabot-preview-hello-dependabot/.

[68] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot", *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4004–4022, 2023. DOI: 10.1109/TSE.2023.3278129.

[69] Anthropic, *Claude sonnet 4*, https://www.anthropic.com/claude/sonnet, Accessed: 2025-06-02, 2025.

[70] *Myflavoria*, https://myflavoria.fi, Progressive Web App used in Flavoria's research platform.

[71] University of Turku, *Flavoria® research platform*, https://www.flavoria.fi/, Accessed: 2025-06-02, 2025.

[72] *Cursor agent*, https://docs.cursor.com/chat/agent, Accessed June 2025, 2025.

Appendix A Supplementary Scripts

Listing A.1: Script to generate SonarQube issues JSON

```
import requests
import json
import math

# Configuration
SONARQUBE_URL = "http://localhost:9000"  # Replace with your
   SonarQube URL
PROJECT_KEY = "MyFlavoria-Master"  # Replace with your project
    key
API_TOKEN = "squ_102c2a42268f9e2763ea64a8ca64fef39bc5c77c"  #
   Replace with your User Token
OUTPUT_FILE = "sonarqube_issues.json"

def fetch_rule_details(rule_key, auth):
    """Fetch rule details to check for remediation suggestions
       ."""
    rule_url = f"{SONARQUBE_URL}/api/rules/show"
    params = {"key": rule_key}
    try:
```

```
        response = requests.get(rule_url, params=params, auth=
            auth)
        response.raise_for_status()
        rule_data = response.json()
        rule = rule_data.get("rule", {})
        # Extract remediation message or description if
            available
        return rule.get("htmlDesc", "").strip() or rule.get("
            name", "")
    except requests.RequestException as e:
        print(f"Error fetching rule {rule_key}: {e}")
        return ""

def fetch_all_issues():
    """Fetch all issues from SonarQube with pagination."""
    issues = []
    page = 1
    page_size = 500  # Max page size allowed by SonarQube
    auth = (API_TOKEN, "")

    while True:
        url = f"{SONARQUBE_URL}/api/issues/search"
        params = {
            "components": PROJECT_KEY,
            "issueStatuses": "OPEN",  # FALSE_POSITIVE (for
                Fetching false positive flagged issues)
            "ps": page_size,
            "p": page
        }
```

```
        try:
            response = requests.get(url, params=params, auth=
                auth)
            response.raise_for_status()
            data = response.json()

            # Extract issues from the response
            page_issues = data.get("issues", [])
            if not page_issues:
                break  # No more issues to fetch

            # Process each issue
            for issue in page_issues:
                raw_file = issue.get("component", "")
                clean_file = raw_file.replace(f"{PROJECT_KEY}:
                    ", "") if raw_file.startswith(f"{
                    PROJECT_KEY}:") else raw_file
                issue_data = {
                    "key":issue.get("key", ""),
                    "description": issue.get("message", ""),
                    "line": issue.get("line", None),
                    "file": clean_file,
                    "severity": issue.get("severity", ""),
                    "type": issue.get("type", ""),
                    "rule": issue.get("rule", ""),
                    "suggestion": ""
                }
```

```
                # Fetch rule details for potential fix
                    suggestion
                if issue.get("rule"):
                    suggestion = fetch_rule_details(issue["
                        rule"], auth)
                    issue_data["suggestion"] = suggestion if
                        suggestion else "No specific fix
                        suggestion available."

                issues.append(issue_data)

            # Check pagination
            total_issues = data.get("total", 0)
            total_pages = math.ceil(total_issues / page_size)
            print(f"Fetched page {page}/{total_pages} ({len(
                page_issues)} issues)")

            if page >= total_pages:
                break  # All pages fetched

            page += 1

        except requests.RequestException as e:
            print(f"Error fetching page {page}: {e}")
            break

    return issues

def save_to_json(issues):
```

```
    """Save issues to a JSON file."""
    output_data = {
        "project": PROJECT_KEY,
        "total_issues": len(issues),
        "issues": issues
    }
    try:
        with open(OUTPUT_FILE, "w", encoding="utf-8") as f:
            json.dump(output_data, f, indent=2, ensure_ascii=
               False)
        print(f"Saved {len(issues)} issues to {OUTPUT_FILE}")
    except IOError as e:
        print(f"Error saving to file: {e}")


def main():
    print("Fetching issues from SonarQube...")
    issues = fetch_all_issues()
    if issues:
        save_to_json(issues)
    else:
        print("No issues found or an error occurred.")


if __name__ == "__main__":
    main()
```

Listing A.2: Script to combine SonarQube and Snyk issues

```
import json


def load_issues(file_path):
```

```
    """Load issues from a JSON file."""
    try:
        with open(file_path, "r") as f:
            data = json.load(f)
            return data.get("issues", [])
    except FileNotFoundError:
        print(f"File {file_path} not found")
        return []
    except json.JSONDecodeError:
        print(f"Error parsing {file_path}")
        return []

def combine_all_issues(sonarqube_file, snyk_file, output_file)
    :
    """Combine all issues from SonarQube and Snyk JSON files
       without limits."""
    sonarqube_issues = load_issues(sonarqube_file)
    snyk_issues = load_issues(snyk_file)

    combined_issues = sonarqube_issues + snyk_issues

    with open(output_file, "w") as f:
        json.dump({"issues": combined_issues}, f, indent=2)
    print(f"Combined all issues written to {output_file} ({len
       (combined_issues)} issues)")

if __name__ == "__main__":
    sonarqube_file = input("Enter SonarQube JSON file path: ")
    snyk_file = input("Enter Snyk JSON file path: ")
```

```
    output_file = "combined_issues.json"

    combine_all_issues(sonarqube_file, snyk_file, output_file)
```