DEVELOPMENT OF AN INTELLIGENT CODE REVIEW TOOL

BY

EKEKWE, TANYALOUISE KELECHI CHISOM
(20CG028072)

A PROJECT SUBMITTED TO THE DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES, COLLEGE OF SCIENCE AND TECHNOLOGY, COVENANT UNIVERSITY OTA, OGUN STATE.

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE BACHELOR OF SCIENCE (HONOURS) DEGREE IN COMPUTER SCIENCE.

JULY, 2024

CERTIFICATION

I hereby certify that this project was carried out by Tanyalouise Kelechi Chisom EKEKWE in the Department of Computer and Information Sciences, College of Science and Technology, Covenant University, Ogun State, Nigeria, under my supervision.

Dr. Itunuoluwa E. Isewon _____
Supervisor **Signature and Date**

Prof. Olufunke O. Oladipupo _____
Head of Department **Signature and Date**

DEDICATION

I dedicate this work to God, who has been my very present help in times of need during my 4-year journey in this institution. I also dedicate this work to my friends and family, who support me.

ACKNOWLEDGEMENTS

My most profound gratitude goes to Almighty God, who has kept me and sustained me right from the very start of my degree up until this point. To Him be all the glory.

I want to express my sincere gratitude to my younger sister, Daniella Ekekwe, who encourages me to follow my passion for computer science, especially during this project.

I am extremely grateful to my parents, Mr. and Mrs. Ekekwe for their sacrifices, contributions, and prayers towards the completion of this degree. I am extremely privileged to have them in my life.

My sincere gratitude also goes to my hardworking supervisor and co-supervisor, Dr. Itunuoluwa Isewon and Ms. Nathaniel Jemimah, who continuously provided their support throughout this project duration. Thank you for your excellent supervision and guidance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AI: Artificial Intelligence
API: Application Programming Interface
AST: Abstract Syntax Tree
CI/CD: Continuous Integration/Continuous Deployment
CNN: Convolutional Neural Network
CSS: Cascading Style Sheet
DCR: DeepCodeReviewer
DQN: Deep Q-Networks
GGUF: GPT-Generated Unified Format
GPU: Graphics Processing Unit
JSON: JavaScript Object Notation
JWT: JSON Web Token
LLM: Large Language Model
MCR: Modern Code Review
MCR: Modern Code Review
NLP: Natural Language Processing
RAM: Random Access Memory
RTK: Redux React ToolKit
SCA: Software Composition Analysis
SVM: Support Vector Machines
TD: Technical Debt

ABSTRACT

This project introduces an intelligent code review tool designed to enhance code quality and enforce programming conventions across different skill levels. The issue of poor code quality, stemming from the neglect of coding standards, presents significant challenges in software development, leading to increased technical debt and maintenance difficulties. To address this problem, the proposed tool leverages a pretrained Large Language Model (LLM), CodeLlama, and static analysis techniques to dynamically identify and provide feedback on code violations. The LLM offers contextual explanations, helping developers understand the rationale behind coding standards.

The methodology involved integrating the LLM with a static analysis engine, creating an intuitive user interface using React and Tailwind CSS, and ensuring efficient state management with React Redux Toolkit and Query. The backend was developed using Express.js and Python's Flask framework to manage API endpoints and interact with the LLM. User feedback was collected through interviews, focusing on metrics such as user experience, interface intuitiveness, and feedback clarity. Results from the evaluation indicate that users found the tool intuitive and its feedback helpful and clear. Comparisons with existing tools, such as FrenchPress, showed that the intelligent code review tool offers a broader scope of code issues and dynamic, contextually relevant feedback. Users reported higher satisfaction and perceived the tool as a significant improvement over current code review solution.

CHAPTER ONE
INTRODUCTION

1.1 Background Information

The evolution of coding standards and best practices has deep roots, dating back to the early days of programming (Dijkstra *et al.*, 1972). As software development methodologies and technologies have advanced, so have coding standards. This co-evolution is particularly evident when examining the shift from structured programming in the 1960s to object-oriented programming in the 1980s (Pratap, 2023). Early coding standards or programming conventions emerged alongside structured programming to promote modular, readable code and followed a clear control flow; these characteristics became essential as software projects grew in complexity (Oviedo, 1984). Structured programming approaches like top-down design and control flow statements (if statements, loops) rely on well formatted code to be easily understandable and maintainable (Kostyrko *et al.*, 2023). Object-oriented design principles like encapsulation and code reusability are best achieved through consistent naming conventions and code organization.

Among software engineering techniques, programming conventions play a critical role in promoting the readability of source code (Oliveira *et al.*, 2020). These conventions, which can encompass aspects like indentation, naming conventions, and code formatting, act as a shared language for developers, allowing them to quickly grasp the purpose and structure of the code, even if written by someone else (Chen *et al.*, 2018). According to Kourie & Pieterse (2008a), there is a need to formally educate students to become professional, responsible, and reliable developers capable of producing quality code. However, Wiese *et al.* (2017) identifies a gap in the existing literature directed at teaching programming conventions during the initial stages of programming education. A concerning find, as poorly formatted and inconsistently named code becomes a significant hurdle as projects grow in size and complexity, hindering both understanding and maintenance. If early adoption of programming conventions is prioritized, beginner programmers can lay a solid foundation of clean, maintainable, and collaborative code throughout their software development journey (Keuning *et al.*, 2021).

One potential solution to bridge this gap and encourage early adoption of programming conventions is through intelligent code review tools (Kim *et al.*, 2022). These tools are already widely used in professional software development to automate tasks like identifying

syntax errors and potential bugs (Sadowski *et al.*, 2018). In the context of learning conventions, these tools can be leveraged as a valuable learning resource for beginners. According to Bancroft & Roe (2006), feedback or recommendations are essential in the learning process, especially when they are available on request. Valuable feedback that can improve proper understanding of programming conventions can be provided by a code reviewing tool that not only highlights potential errors but also flags sections that deviate from established coding standards. This real time feedback can raise beginners' awareness of the importance of proper formatting, naming conventions, and code organization. By integrating educational resources and suggestions directly into the development workflow, intelligent code reviewing tools can become powerful allies for beginner programmers, helping them write clean, maintainable, and well-structured code from the very beginning.

While established code reviewing platforms like those offered by GitHub and Codacy are valuable tools in the programmer's arsenal, they have a blind spot, programming conventions. These platforms primarily focus on identifying functional errors and bugs, ensuring the code works as intended (Badampudi *et al.*, 2023). However, they often fall short in addressing whether the code adheres to established programming conventions. Some might offer essential error messages related to "code smells" or stylistic inconsistencies but lack the depth and interactivity needed to be a true learning resource for beginners who need to understand the "why" behind coding conventions (Wessel *et al.*, 2020). This gap in feedback leaves aspiring programmers vulnerable to developing bad habits and writing code that becomes a tangled mess down the line.

Intelligent code reviewing tools, designed specifically for beginners, can address this limitation by transforming code review into a powerful learning resource. These tools leverage AI and established coding standards to go beyond simply highlighting missed conventions (Kim *et al.*, 2022). This study proposes a real-time mentor who does not just point out formatting issues (a language linter) but also suggests best practices and alternative approaches based on the specific code and its intended function. The shift towards a more interactive and personalized learning experience could significantly improve the quality, maintainability, and understanding of code written by beginners (Shen & Lee, 2020). Ultimately, this will lead to a smoother transition into professional software development by equipping them with the necessary skills to write clean, maintainable, and collaborative code from the outset.

1.2 Statement of the Problem

In the early stages of learning how to program, much attention is focused on how to implement basic functionality and compose a running program; thus, coding standards are usually disregarded. As noted by Kirk *et al.* (2020), there is a considerable gap in the instruction of code quality: 30% of introductory university programming courses include it. It is for this very reason that novice programmers are often unable to become very sensitive to code quality. Understandably, new programmers are eager to understand the basic functionalities of languages and develop programs that work, often neglecting to apply coding conventions during their development workflow (Kohlbacher *et al.*, 2023). According to research conducted by Joni & Soloway (1986), 90% of students used the wrong coding standards in programming exercises, and Ruvo *et al.* (2018) finds that these bad habits persist up till into students' 4th year of a BS degree. New coders often avoid learning conventional methods; however, disregarding coding standards and conventions encourages bad coding habits that affect code readability, maintainability, and scalability in the long run (Popić *et al.*, 2018).

The effects of poor code quality don't stop at the developers themselves. It adversely affects the software industry as a whole, by introducing a concept known as technical debt (Han *et al.*, 2020). Technical debt is a metaphor that refers to the hidden costs of neglecting good coding practices. Just as financial debt accrues interest over time, poorly written code accumulates complexity and becomes harder to understand and modify as the project grows (Digkas *et al.*, 2022). Poor code quality leads to problems such as higher costs as bug fixes and new features become cumbersome and an increased risk of software failure due to vulnerabilities.

Although existing code review platforms offer insights into basic coding style issues, there's an opportunity to enhance their capabilities for beginners, particularly by addressing the gap in feedback on the "why" behind conventions (Wessel *et al.*, 2020). Understanding the rationale is crucial for preventing bad habits and tangled codes that create future maintenance and collaboration challenges. This highlights the need to integrate coding conventions into the learning process better. This study addresses this need by developing an intelligent code-reviewing tool with a tutoring focus. A novel approach that has the potential to significantly improve the overall quality of code produced in our industry.

1.3 Aims and Objectives

The aim of this study is to develop an intelligent code-reviewing tool designed with a tutoring focus to empower beginner programmers in understanding and applying coding

conventions using a pretrained Large Language Model and simple static code analysis. The objectives of this study are:

(i) To identify the specific challenges beginner programmers face in learning programming conventions.

(ii) To gather and process the coding styles and general standards data for the static code analysers to identify potential style violations based on those datasets.

(iii) To design and develop the intelligent explanation module and the interactive guidance features of the code reviewing tool.

(iv) To design the user interface (UI) and user experience (UX) of the intelligent code review tool.

(v) To implement a working prototype of the intelligent code review tool.

1.4 Methodology

(i) Review existing literature on programming conventions and analyse and observe existing systems.

(ii) Acquiring sufficient datasets from coding standards documents or open-source code repositories and processing them using data processing tools such as Python libraries (Pandas, Natural Language Toolkit (NLTK), or Spacy).

(iii) Modelling and designing the intelligent code review tool components using UML diagrams such as an activity diagram, sequence diagram, use case diagram, etc., and system architecture diagrams.

(iv) Designing the user interface (UI) and user experience (UX) of the tool using the design tool Figma.

(v) Implementation of a working prototype of the tool using the React frame- work for the frontend, Rust for the static analyser component, and for the backend and transfer learning with the CodeLlama-7B-Instruct-GGUF Model based off of Meta's CodeLlama 7b Instruct (Rozière *et al.*, 2023a) for the explanation module.

Table 1.1: Objectives Methodology Mapping Table

S/N	Objectives	Methodology
1	To identify the specific challenges faced by beginner programmers in learning programming conventions.	**Extensive Literature Review and Existing Systems** Review existing literature related to beginner developers and programming

S/N	Objectives	Methodology
		conventions and Analyze and observe existing systems.
2	To design and develop the intelligent explanation module and the interactive guidance features of the code reviewing tool.	**Design and Development of Components** Modelling and designing the intelligent code review tool components using UML diagrams such as an activity diagram, sequence diagram, use case diagram, etc., and system architecture diagrams.
3	To design the user interface (UI) and user experience (UX) of the intelligent code review tool.	**UI/UX Design of Components** Designing the user interface (UI) and user experience (UX) of the tool using the design tool Figma.
4	To implement a working prototype of the intelligent code review tool.	**Implementation of a working Prototype** Implementation of a working prototype of the tool using the React framework for the frontend, Redux Toolkit (RTK) Query for the Git Authentication and API endpoints, Rust for the static analyser component, and Node.JS for the backend. Implementing the explanations module using CodeLlama-7B-Instruct-GGUF Model based off of Meta's CodeLlama 7b Instruct (Rozière *et al.*, 2023a).

1.5 Significance of the Study

This project suggests a new intelligent code reviewing tool that tackles the absence of beginner friendly learning materials for beginners on conformity to syntax. First, through prompt feedback, clear explanations and counselling that is interactive, this tool enables novices to produce neat code which could present an entirely different method regarding teaching programmers via open source LLMs as well as interactive functions.

The benefits of this project extend beyond individual programmers as the integration of the proposed tool in teaching can result to cleaner and more maintainable code in the software development industry down the line. This improvement in code will also translate into

significant decrease in cost for businesses. Additionally, fostering a common understanding of coding conventions can enhance collaboration and communication among developers, leading to more efficient code reviews and faster development cycles.

Most importantly, a proper grasp of coding conventions by new programmers can minimize the time and resources to be invested in orienting them for new projects. Generally speaking, this project is likely to have a very high impact on the software development industry because it empowers the programmer; it may spur innovative education and, at last, contribute to building an environment of development that is more efficient and productive.

1.6 Scope of the Study

This project maintains a strict development focus to address the gap in learning resources concerning programming conventions for beginner programmers. The intelligent code reviewing tool being developed will leverage existing open-source large language models trained on code to provide explanations and justifications.

The Interface of the system is designed to be easy and intuitive for new users. It contains interactive elements, such as step-by-step guidance through tutorials or visual guidance, all the way to the point where code refactoring recommendations are made to help them understand code violations they make in their code. This work targets the enhancement of existing open-source, large language models (LLMs) with quick engineering solutions, with no need to conduct user studies or create new LLM architectures.

While it does not include user research or the creation of entirely new LLM models, the project still intends to make existing pretrained open-source LLMs more effective by utilising prompt engineering techniques. It will involve development during the 16 weeks, and its target audience will be programmers at the beginner level who have some knowledge of JavaScript. With this scope, we make it clear that the project is focused on the task of development but, at the same time, serves a beneficial objective—providing a handy tool for empowering beginner-level programmers to improve their coding skills.

1.7 Limitations of the Study

The major limitations of this study include the following:

(i) Limited Language Support: Initially, the tool will only be able to review code written in one programming language, JavaScript, limiting its reach to programmers using that specific language.

(ii) Pretrained Model Dependency: The quality of the explanations and justifications provided by the LLM component is heavily dependent on the data used during its pretraining. If the pretrained model has been exposed to limited or biased data, it could result in inaccurate or misleading responses.

(iii) Limited Scope of Conventions: The first iteration of the tool will focus on a core set of coding conventions. We plan to expand the scope to address a wider range of conventions in future versions.

1.8 Structure of the Research

Chapter One of the project contains an explanation of the project, problems with existing solutions, the need for an improved solution, the method of implementation, the significance of the study, and the limitations. Chapter Two describes the existing systems related to the project topic, the methodology, algorithm, and techniques used in related systems. Chapter Three describes the analysis and system design. Chapter Four shows the implementation of the system in detail and the results obtained. Chapter Five summarizes the project and gives recommendations, suggestions, conclusions, and references.

CHAPTER TWO
LITERATURE REVIEW

2.1 Preamble

The importance of coding conventions remains a hurdle for beginners in today's software development landscape, particularly when it comes to ensuring clean and well-formatted code. This literature review lays the groundwork for a solution: an intelligent code review system utilizing open-source, code-focused LLMs and static analysis. This literature encompasses the review of programming conventions or coding standards, coding smells and code quality, code quality management, lack of programming conventions and its effects

2.2 Review of Programming Conventions

Programming conventions, also known as coding conventions or coding standards, are repositories of rules and guidelines that encompass all aspects of improving code quality (Smit *et al.*, 2011). These guidelines within software development define borders, recommend certain methodologies or impose limits affecting designers', developers' and maintainers' approach to software systems (Rodrigues & Montecchi, 2019).

However, programming conventions are not static. As programming languages evolve and new paradigms emerge, these conventions co evolve to influence how code is structured, written, and maintained. For example, the transition from procedural languages, C, to object oriented languages, C++ and Java, introduced entirely new conventions related to class structures, inheritance, and polymorphism (Pressman & Maxim, 2014).

2.2.1 Brief History of Programming Conventions

In the early days, 1940s 1960s, of languages like FORTRAN and COBOL, conventions focused primarily on basic formatting and readability due to limited processing power and the linear nature of these languages (Medoff, 2015). Clear documentation was crucial for maintaining code written on punch cards, so conventions often encouraged detailed comments and code structure mirroring the documentation layout.

With the rise of procedural programming languages like C in the 1960s 1980s, conventions evolved to address modularity and control flow. Indentation became a cornerstone for defining code blocks and functions, promoting better organization and readability (Kernighan & Ritchie, 1978). As code sharing and collaboration increased, platform specific

conventions emerged to ensure portability and maintainability across different systems (Pressman & Maxim, 2014).

The shift to object oriented languages like C++ and Java in the 1980s 2000s necessitated entirely new conventions for class structures, inheritance, and polymorphism (Meyer, 1997). Community driven coding standards like PEP 8 for Python, (van Rossum *et al.*, 2001), or Google's JavaScript Style Guide further emphasized code consistency within specific programming communities (Google, n.d.). These trends continue in the modern age, with conventions focusing on maintainability and scalability in complex, distributed software projects (Beck, 1999). Integration with code analysis tools further enforces adherence to conventions, ensuring clean, maintainable code remains a priority regardless of the programming language or paradigm used.

2.2.2 Main Types of Programming Conventions

There three main types of coding conventions which include: Naming conventions, Formatting conventions and commenting conventions. In programming, naming conventions establish guidelines for naming variables, functions, classes, and other code elements. These conventions promote consistency, clarity, and readability throughout the codebase (Herka, 2022). Following them allows developers to readily grasp the purpose and role of different code elements. For instance, using descriptive names like calculateTotalPrice instead of cryptic abbreviations enhances code understandability and maintainability (Pugh, 2018). Below, in figure 2.1, are some examples of naming conventions in Python:

```python
# Variable and function names in lowercase_with_underscores
total_count = 0
calculate_total()

# Class names in CapitalizedCamelCase
class UserAccount:
    pass

# Constant names in ALL_CAPS
PI = 3.14159
```

Figure 2.1: Some naming conventions in python

Formatting conventions encompass practices related to the visual structure of code. These conventions dictate aspects such as indentation, spacing, line breaks, and the use of code blocks (Piater, 2005). Adhering to formatting standards enhances code maintainability

and comprehension. Consistent formatting makes code easier to read, debug, and modify, especially when multiple developers collaborate on a project .

Commenting conventions provide guidelines for writing clear, concise, and informative comments within the codebase (Huang *et al.*, 2018). Comments are essential for explaining the functionality and intent of the code, especially in complex or critical sections. Huang *et al.* (2018) highlights challenges faced by developers in sticking to using commenting conventions in their development processes. He proposes a tool, CommentSuggester, to guide developers in making informed commenting decisions. Established from various sources, effective comments help developers understand the reasoning behind specific code decisions, making it easier to maintain and update the code in the future proposes a method (Haouari *et al.*, 2011; Rani *et al.*, 2021).

2.3 Review on Coding Smells and Code Quality

In software development, code quality plays a crucial role in determining the reliability, maintainability, and efficiency of a software system. While robust code ensures a program's reliability and efficiency, clear and readable code simplifies future modifications and updates. High quality code not only enhances the functionality and performance of an application but also simplifies the process of future modifications and updates.

Code quality is a concept that lacks a precise definition in the literature, often characterized by vague and varied interpretations (Keuning *et al.*, 2023). To comprehensively assess or categorize code quality, it is essential to consider the static properties of the code as opposed to the dynamic properties of code such as correctness, test coverage, and runtime performance. The evaluation of these static properties, as outlined by Stegeman *et al.* (2016) encompasses criteria such as documentation, layout, naming, flow, expressions, idiom, decomposition, and modularization. Adherence to coding standards is essential across all these categories to ensure a high level of code quality.

However, problems found in these categories (Stegeman *et al.*, 2016), are called "code smells". This term was introduced by Fowler, (1999). Code smells might hint at deeper problems in the design of functionally correct code, affecting the quality of code (Albuquerque *et al.*, 2023; Tufano *et al.*, 2015). These smells can be a consequence of prioritizing short term goals like reusability over long term maintainability. While reusability is desirable for well crafted software (Pandey *et al.*, 2020), pressures like frequent requirement changes, increasing project size, and time constraints can lead to code deterioration (Verma *et al.*, 2023).

One crucial aspect of code quality lies in the inherent trade offs developers face when optimizing different code characteristics. For instance, prioritizing code readability and understandability might come at the expense of achieving peak performance (Sas & Avgeriou, 2020). Conversely, highly optimized code for speed might be less maintainable in the long run, requiring more effort for future modifications.

2.4 Review on Code Quality Management

Code quality management refers to the systematic process of overseeing and ensuring the quality of code within a software development project. It involves implementing strategies, practices, and tools to maintain high standards of reliability, maintainability, and efficiency in the codebase (Plosch *et al.*, 2010).

For a long time, peer code review, a manual inspection of code by other developers on a software development team, has been recognised at a tool for improving the quality of code (Sadowski *et al.*, 2018). Through peer code review, developers can identify potential issues like bugs, code smells, and suboptimal coding practices. This collaborative approach fosters knowledge sharing, improves coding consistency, and ultimately elevates the overall quality of the codebase.

2.4.1 Modern Code Review: A Streamlined Approach

However, the limitations of manual code review or peer review in today's fast paced development environments are becoming increasingly apparent. The sheer volume of code produced, coupled with time constraints, can make thorough code review a challenge (Sadowski *et al.*, 2018). The immediate solution adopted today to this limitation is called Mordern Code Review (MCR).

MCR is a lightweight approach to traditional code inspections or peer review (Badampudi *et al.*, 2023). During the process, one or more reviewers assess the code for errors, adherence to coding standards, test coverage, and more. These MCR tools automate many aspects of the review process, making it faster and more efficient compared to traditional peer review methods that rely solely on manual processes (Qiao *et al.*, 2024). For example, tools like GitHub, GitLab, and provide features such as inline commenting, automated code formatting checks, and integration with Continuous Integration/Continuous Deployment (CI/CD) pipelines, streamlining the review process.

MCR's process, as explained in the works by Badampudi *et al.* (2023), follows a series of six main steps, shown in figure 2.2, that integrate with version control systems (e.g., Gerrit, GitHub and GitLab). The steps involved in this process include:

(i) In Step 1, the code author(s) or developers submit code or changes. Usually, version control systems like Gerrit and GitHub are used, the developer creates a pull request.

(ii) In Step 2, the project or code owner selects one or more reviewers, usually using heuristics, to review the new pull requests made by the code author(s).

(iii) In Step 3, the reviewer(s) are notified of their assignment.

(iv) In Step 4, the reviewer(s) check the code for defects or suggest improvement.

(v) In Step 5, the reviewer(s) and author(s) discuss the feedback from the review.

(vi) In Step 6, the pull request or code change is rejected or sent back to the author(s) for refinement. If no further rework is required, the code change is merged into the codebase.

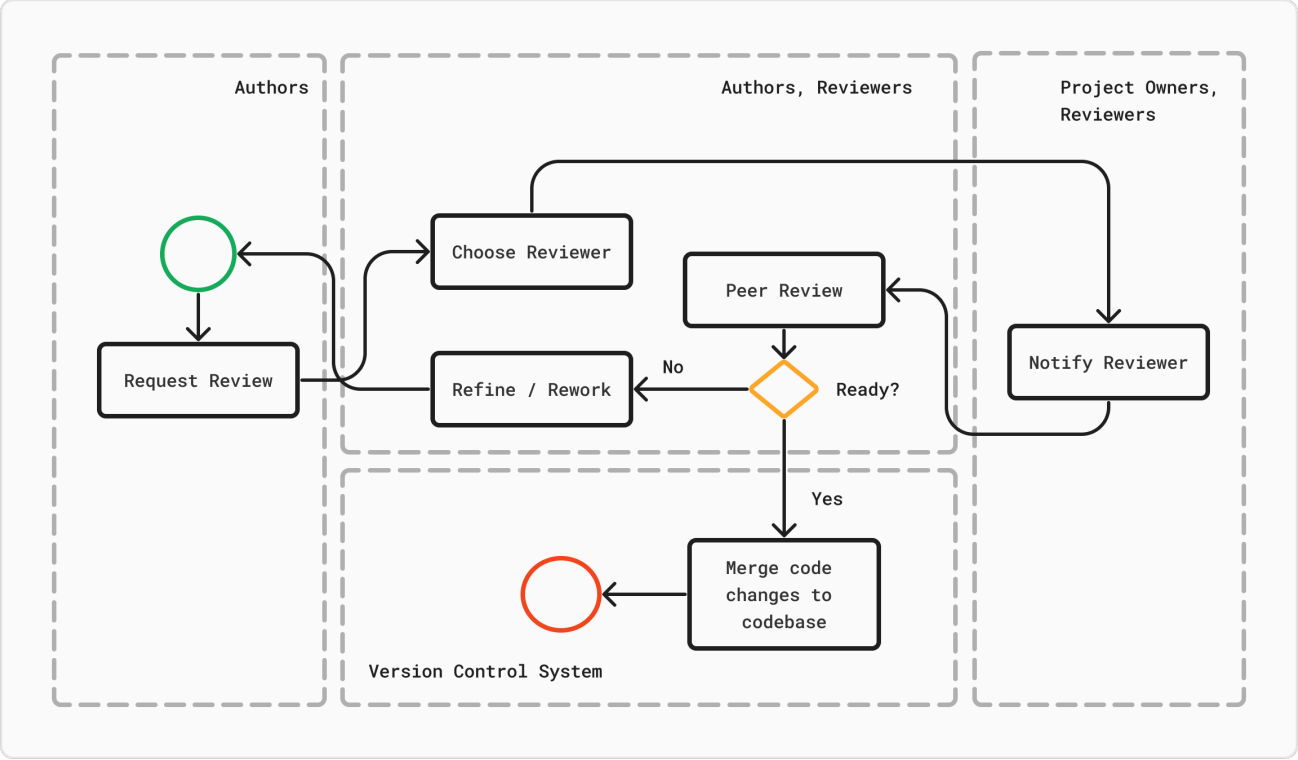

Figure 2.2: Overview of steps in modern code reviews, adapted from Badampudi *et al.* (2023)

2.4.2 Intelligent Code Review Tools

Software development teams using the MCR process struggle to keep pace with the ever increasing scale of software projects (JingKai *et al.*, 2019), as code reviewers often have to allocate a huge amount of time in performing code review tasks due to the amount of code changes in large scale codebases. While some part of the code review processes have been automated with the use of linters or static analysis tools that contain rules related to coding best practices (Bielik *et al.*, 2017), there's still a huge amount of effort put into the review process, as these tools mainly detect common issues and leave important aspects such as defects, architectural issues, and testing for code (Gupta, 2018).

Although defect detection remains important in these code review processe (Fan *et al.*, 2018), industry developers increasingly value code quality aspects that promote long term maintainability, understanding of source code and code improvement (Alberto & Christian, 2013). Many research studies aim to reduce the amount of time spent on code review processes by providing solutions to close the learning gap of these systems. Research work such as DeepCodeReviewer (DCR) by Gupta (2018), aims to recommend reviews by learning the relevancy between the source code and the review. The Code Review Bot, (Kim *et al.*, 2022), is designed to process review requests holistically regardless of such environments and checks various quality assurance items such as potential defects in the code, coding style, test coverage, and open-source license violations.

Essentially, intelligent code review tools, such as DeepCodeReviewer (DCR) and Code Review Bot, use deep learning and automation to enhance the code review process. DCR, for example, recommends relevant code reviews based on historical data, while Code Review Bot checks for potential defects, coding style, test coverage, and open-source license violations. These tools have been found to be effective in improving the efficiency of change based code review. They have also been well received by developers, with positive feedback and active response to reviews.

2.4.3 Static Analysis Tools

Static code analysis tools play a crucial role in software development by assisting developers in enhancing the quality of their code. These tools statically evaluate source code to identify bugs, security vulnerabilities, duplications, and code smells. As noted by Møller & Schwartzbach (2020), the practice of static analysis has also been instrumental in optimizing compilers since the 1960s and has since expanded to aid in software debugging and quality improvement (Danilo *et al.*, 2021). The adoption of static analysis tools has

become increasingly prevalent in the industry, offering developers insights into code quality standards and potential defects (Ilyas & Elkhalifa, 2016).

There are various types of static analysis tools available, each specializing in different programming languages and types of defects. Some tools focus on general purpose code analysis, while others are tailored for specific languages like C/C++, Java, or Python. These tools such as Cppcheck, FindBugs, and SonarQube, use predefined rules or patterns to identify issues and provide actionable insights to developers for improving code quality. Static analysis tools are continuously evolving to meet the changing needs of software development and to provide more comprehensive coverage in detecting defects and vulnerabilities within the codebase (Nachtigall *et al.*, 2019).

2.5 Review on the Effects of Unclean Code

"Unclean code" is a term used to describe source code that is difficult to maintain, evolve, and change, often due to poor software engineering practices (Carvalho *et al.*, 2017). This type of code is characterized by poor readability, lack of maintainability, and the presence of "code smells" - indicators of software design problems (Gupta, 2018). These code smells, like long and complex functions scattered throughout the codebase (making them harder to identify and manage) can significantly increase the likelihood of errors being introduced during modifications, change proneness, and faults remaining undetected, fault-proneness (Palomba *et al.*, 2017). As a result, unclean code can lead to a vicious cycle of bugs, rework, and project delays, ultimately impacting software quality and project success.

While the importance of clean code in preventing resource loss has been well established (Digkas *et al.*, 2022), unclean code remains a persistent challenge in real world software development projects (Chirvase *et al.*, 2021). Several factors contribute to this issue, including a lack of developer motivation, limited knowledge of clean code principles, and a general lack of awareness regarding the importance of code quality. Inappropriate code reuse practices, often fuelled by time pressures and influenced by the ethical climate within a development team (Sojer *et al.*, 2014), further exacerbate the problem. To address this challenge, it's crucial to encourage developers to prioritize clean code practices and reassess their professional values in relation to their craft.

2.5.1 Technical Debt

Technical debt (TD) is a metaphor that represents the compromise between maintaining clean, well structured code and rapid development (Ampatzoglou *et al.*, 2016). These relate

to design choices, development decisions, and coding practices that prioritize immediate convenience over long-term sustainability (Albuquerque *et al.*, 2023). Although technical debt can bring some of the so-called benefits—such as the reduction of time to market—in the longer term, it can damage overall software quality, whether through added bugs, increased complexity, or making future maintenance efforts more difficult (Rios *et al.*, 2019).

In managing technical debt, various strategies are considered for addressing prioritization, refactoring, and automation. The prioritization of the new features against repayment of the debt is made depending on the impacts to the software quality, business value, and risks (Lenarduzzi *et al.*, 2021). Refactoring is restructuring the code without changing the external behavior, with the aim of making it readable, maintainable, and of good quality. Continuous refactoring keeps reducing technical debt and enhancing long term maintainability. Automated code analysis tools like SonarQube and Findbugs assist in identifying indicators of technical debt, such as code smells, duplication, and complexity, that are helpful for effective detection and resolution of debts.

2.6 Review of Relevant Concepts

This section reviews relevant concepts that are essential for understanding the proposed intelligent code review tool. These concepts include Artificial Intelligence, Machine Learning, Convolutional Neural Networks, Natural Languge Processing, Parsing techniques and processes.

2.6.1 Artificial Intelligence

Artificial Intelligence can be termed as the study of intelligent agents that can act on stimulus or cues from their environment (Russell & Norvig, 2016). It is based on machine learning algorithms and technologies, allowing machines to apply cognitive abilities and perform tasks autonomously or semi autonomously. According to Bartneck *et al.* (2021), there are different aspects of artificial intelligence in which researchers focus on creating machines that think like humans, machines that act like humans, machines that act rationally, or machines that think rationally. There are many different types of AI systems utilized in the world today. Systems such as expert systems, neural networks, and genetic algorithms are all examples of AI systems that are used in various applications.

2.6.2 Machine Learning

This is a sub-field of AI focused on the creation of algorithms that enhance their performance on specific tasks by learning from experience and receiving feedback through performance measures (Shaveta, 2023). Typically, this sub-field is further sub-categorised into three types of learning. These types of learning include:

A. Supervised Learning

Supervised Learning is one of the forms of machine learning in which the AI model is trained on a labelled dataset, meaning that each training example is paired with an output label. The labelling process usually involves human effort to review each piece of data and assign appropriate labels to it (Jiang *et al.*, 2020). According to Shaveta (2023), supervised learning can be regarded as a pupil learning under a teacher's supervision. There are many algorithms used in supervised learning, such as Linear Regression, Logistic Regression, Support Vector Machines (SVM), Naïve Bayes etc. The Naïve Bayes algorithm is a probabilistic classifier based on Bayes' Theorem. This theorem is represented in Equation 2.1 below.

$$\boldsymbol{P\left(\frac{A}{B}\right) = \frac{P\left(\frac{B}{A}\right)P(A)}{P(B)}} \qquad (2.1)$$

Where:

- **(i)** P(A/B): is the posterior probability of class A given the features B.
- **(ii)** P(B/A): is the likelihood of features B given the class A.
- **(iii)** P(A): is the prior probability of class A.
- **(iv)** P(B): is the prior probability of features B.

B. Unsupervised Learning

This is the type of learning in which the machine learns without any human intervention or effort. It involves training the models on data that does not have labelled responses, focusing on the hidden structures or unidentified existing patterns within the data (Alzubi *et al.*, 2018). The figure below shows the steps involved in simple unsupervised learning.

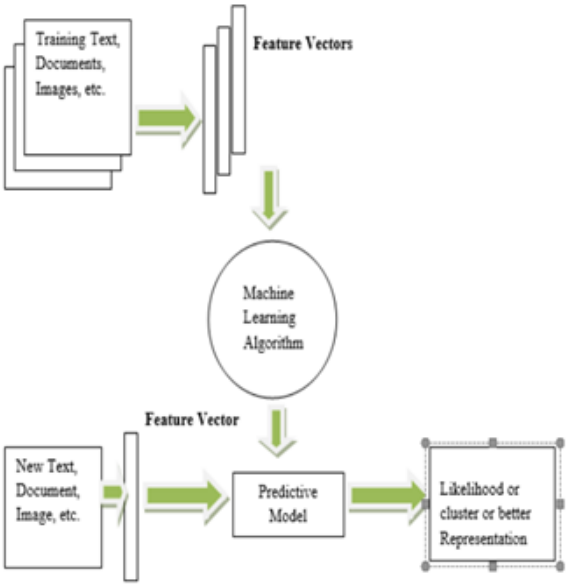

Figure 2.3: Unsupervised Learning [Source: (Alzubi *et al.*, 2018)]

C. Reinforcement Learning

This is the third type of machine learning. This does not focus on the labelling of data, but rather the system receives reward points for each correct action, or a penalty point for each incorrect action. The algorithms explore and rule out various possibilities to get the correct output with the maximum reward. Reinforcement learning, can be described as a technique that can be used to tackle problems involving sequenced decisions and long-term objectives, such as game play or robot navigation (Priyanka, 2024). Common algorithms include Nash-Q Learning, Minimax-Q Learning and Fictitous Play algorithms.

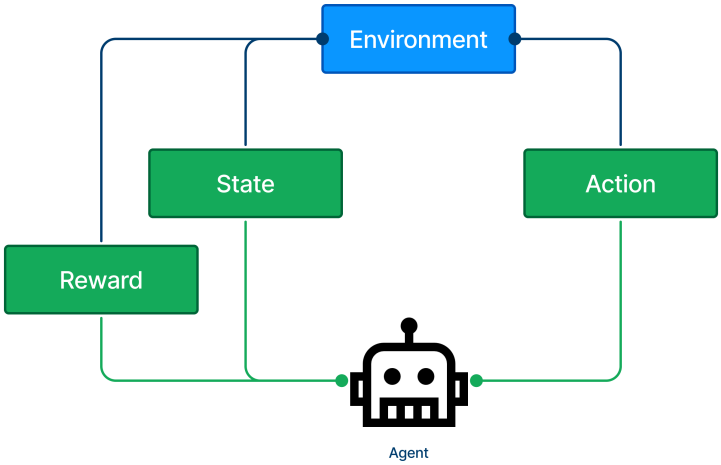

Figure 2.4: Structure of Reinforcement Learning [Source: (Priyanka, 2024)]

2.6.3 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) is a specialized sub-discipline of artificial intelligence engineered to automatically and dynamically learn spatial hierarchies of attributes from input images through a succession of convolutional layers. The structure of CNNs was motivated by neurons in human and animal brains, akin to a standard neural network (Alzubaidi *et al.*, 2021). These networks are particularly effective for tasks involving image and video recognition, classification, and segmentation.

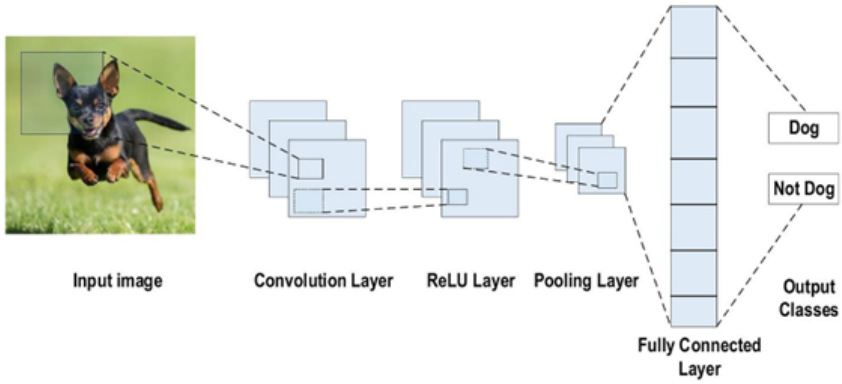

Figure 2.5: An example of CNN architecture for image classification. [Source: (Alzubaidi *et al.*, 2021)]

2.6.4 Natural Language Processing (NLP)

Natural Language Processing (NLP) is another sub-field of artificial intelligence that focuses on the interaction between computers and human language, enabling machines to understand, interpret, and generate human language in a meaningful way (Russell & Norvig, 2016). Within NLP, large language models (LLMs) have emerged as powerful transformative technologies. LLMs, such as OpenAI's GPT-4 and Google's BERT, are trained on vast amounts of text data and leverage advanced neural network architectures, particularly transformers, to capture the nuances of language (Xue, 2024). NLP can be further classified into 2 categories:

(i) Natural Language Understanding (NLU): NLU is a subset of natural language processing in which syntactic and semantic analysis of text and speech is done to make out meanings of sentences. It aids computers in comprehending human language by analyzing and interpreting basic message or speech components (Ovchinnikova, 2012). It is trained with natural user utterances annotated with entities and expanded using synonyms.

(ii) Natural Language Generation (NLG): NLG is another subset of natural language processing. While natural language understanding focuses on computer reading comprehension, natural language generation enables computers to write. NLG is the process of producing a human language text response based on some data input (Perera & Nand, 2017).

2.6.5 Syntax Analysis

Syntax analysis, also known as parsing, is a crucial phase in compiler and interpreter design that checks the correctness of source code structure (Lee, 1982). It involves lexical analysis (scanning) to convert character streams or the source code into tokens, simplifying the parser's job. The parser then generates a parse tree or abstract syntax tree (AST) to represent the code's structure. figure 2.5 shows the workings of a syntax analyzer in a compiler.

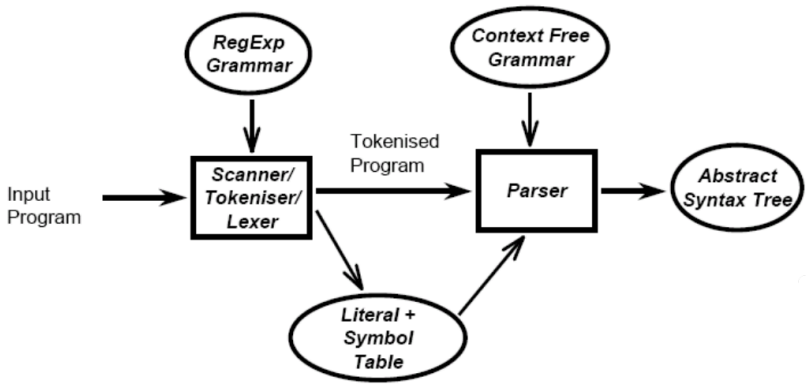

Figure 2.6: Syntax Analysis in a Compiler [Source: (Bhat *et al.*, 2022)]

A. Tokenization: Lexical Analysis

Tokenization or Lexical Analysis is the intial stage of compilation which involves breaking down a stream of code into smaller, meaningful units called tokens (Farhanaaz & Sanju, 2016; Hippisley, 1976). These tokens can be keywords (like if, for, or while), identifiers (user defined names like variable or function names), operators (such as +, , or *), or special symbols (like ;, (, or)). This process of tokenization is analogous to how we segment sentences into individual words. Just as spaces separate words in a sentence, specific delimiters in the code (like whitespace, punctuation, or operators) separate between tokens in the code stream. The importance of tokenization in static analysis stems from its role in transforming raw code into a structured representation, enabling the parser to understand the code's structure and meaning. figure 2.7 shows an overview of the tokenization process.

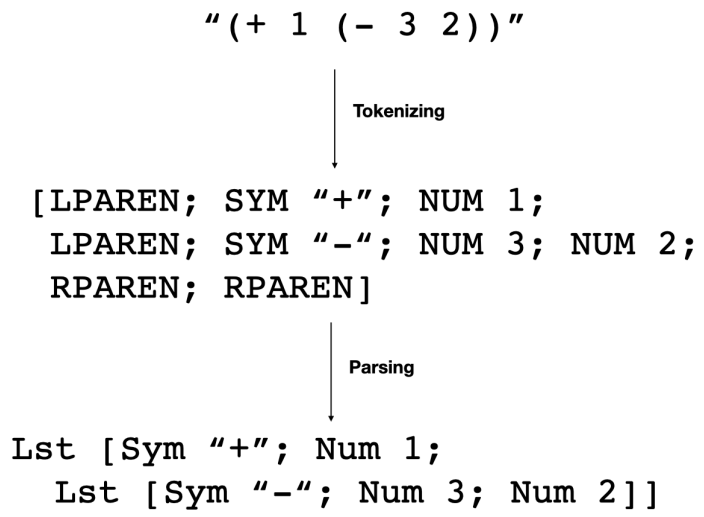

Figure 2.7: Tokenization and parsing overview

B. Parsing

Parsing is a fundamental concept in computer science, particularly in programming languages and natural language processing (Zaytsev & Bagge, 2014). It refers to the process of examining a sequence of symbols (tokens) to ensure its structure and meaning adheres to the set rules of the grammar of a particular language. It can be described as taking a jumbled mix of words, a sentence, and rearranging the words to form a grammatically correct and comprehensible sentence. Parsing is sometimes called the syntax analysis stage of a compiler, as it involves analyzing the syntax of a language to determine its structure (Bhat *et al.*, 2022). figure 2.6 Shows an example of a set of production rules or the grammar of a language.

```
  <alpha>  := 'a' | 'b' | 'c' | 'd' | 'e' | /* ... */ 'z'
            | 'A' | 'B' | 'C' | 'D' | 'E' | /* ... */ 'Z'
<digit>  := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Decl>   := 'var' <Idents> '\n'? ';'
<Idents> := <Ident> ('\n'? ',' <Ident>)*
<Ident>  := <alpha>+ (<alpha> | <digit> | '_')*
```

Figure 2.8: EBNF: An example of a grammar for a programming language

The second phase of a compiler is the parsing phase, in which the source code that has been tokenized, is analyzed by the parser using the content of the production rules in the grammar of the language. The output of this stage is a parse tree or an Abstract Syntax Tree (AST). This AST serves as a tree like representation of the code's structure, capturing the

hierarchical relationships between different code elements. The AST essentially becomes a blueprint of the code, reflecting its organization and flow without including concrete details like variable names or specific values. An example of an AST is shown in the figure below.

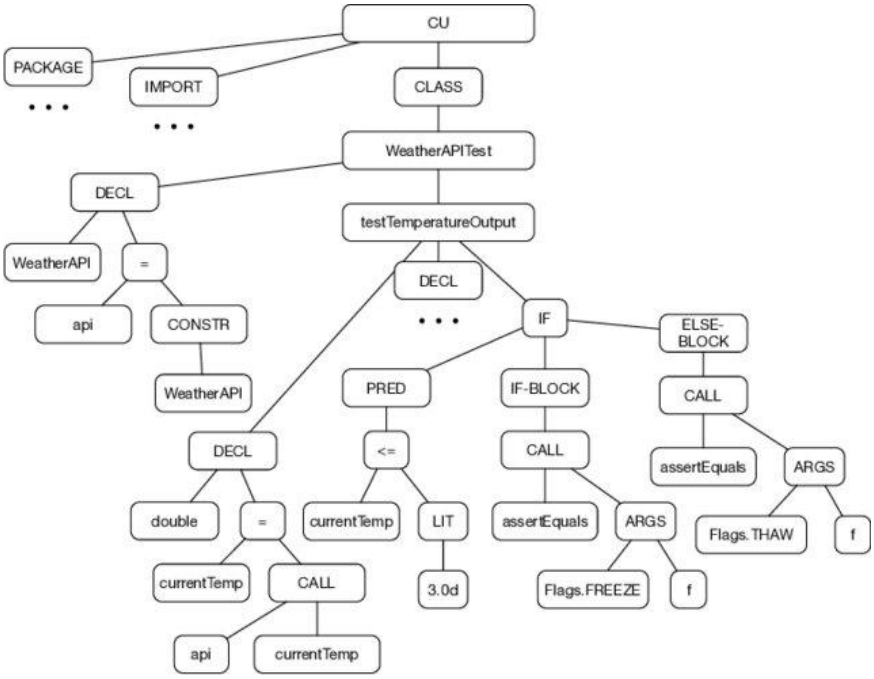

Figure 2.9: A simplified abstract syntax tree (AST)

2.7 Review of Relevant Systems

The review of relevant systems in this paper aims to explore various tools and approaches that have been developed to enhance code quality and improve the code review process, especially for beginners in programming. These systems include:

2.7.1 FrenchPress (Blau & Moss, 2015)

Blau & Moss (2015) proposed a simple IDE plug-in, that aims to provide automated feedback and explanations on flaws or violations found in Java programs, i.e. focusing on diagnosing silent flaws that do not trigger compile-time or runtime errors. This solution is specifically aimed at intermediate students learning their second or third Java course.

Once the plug-in is integrated with the Eclipse IDE, FrenchPress analyses the student's Java code and identifies specific flaws built into the system. Some of these flaws include, field could have been a local variable, non-static method declared public and redundant Boolean expressions. After identifying these flaws, the plug-in generates explanations and is then shown to the user or student within the Eclipse IDE as shown in figure 2.10, allowing the user to see the feedback in the context of their code. The student can then review the feedback and accept the suggestions or reject them. In the study, the tool's ability to motivate

students to accept the changes varied with 36% to 64% of students choosing to modify their programs based on the feedback.

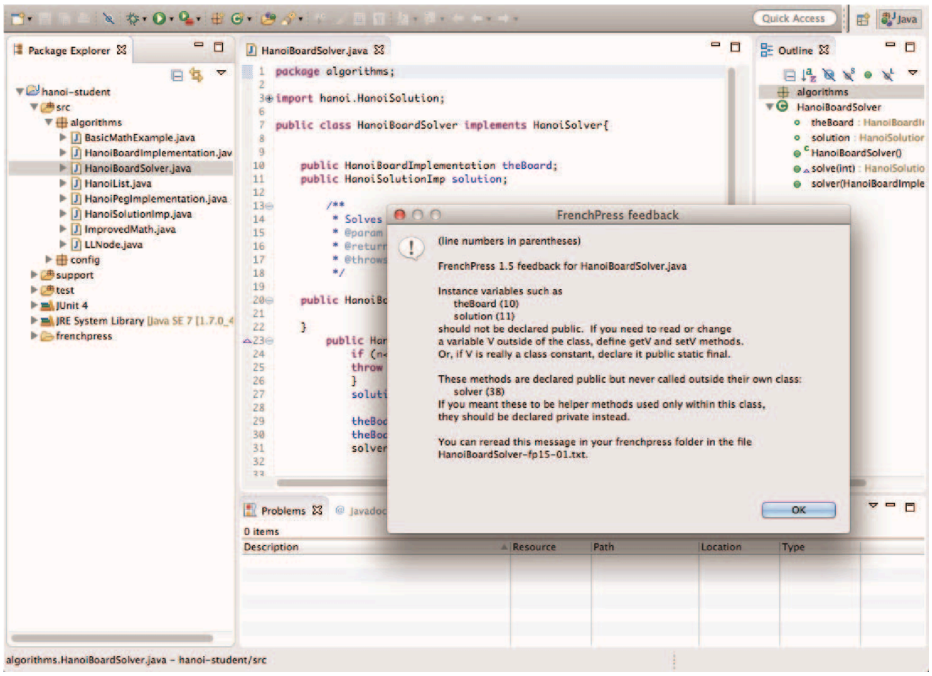

Figure 2.10: FrenchPress feedback [Source: (Blau & Moss, 2015)]

A. Limitations of FrenchPress and How the Proposed Tool Addresses Them

One of the limitations of the FrenchPress would be its narrow target audience. Based on the target audience, the FrenchPress is specifically designed for intermediate Java developers while the intelligent code-review tool offers personalized feedback suitable for a range of skill levels, from beginners to more experienced programmers, making it more inclusive.

Another limitation of FrenchPress is its fixed list of code issues. The plugin uses a limited predefined set of style issues and violations, in its code analysis scope. This inflexibility means that FrenchPress may not cover more range of coding standards and practices required for comprehensive code quality assessment. The proposed intelligent code-review tool overcomes this limitation by utilizing a larger set of code violations from standard guides.

Lastly, the proposed intelligent code review system uses a LLM for the feedback or explanation aspect of the system. An advantage over the plugin in which all the feedback is predefined and tailored to the violations. The table below is a comparison of the features of the proposed solution and the FrenchPress plugin.

Table 2.1: Comparison of features between FrenchPress and the proposed Intelligent Code Review Tool

Feature	FrenchPress	Proposed Intelligent Code Review Tool
User Input Options	Only supports local code in Eclipse.	Supports inputting code directly or linking codebases from code hosting sites.
Contextual Explanations	Provides explanations using student-appropriate vocabulary.	Generates detailed, contextually relevant explanations and justifications based on code content.
Review Summary	Not Available	Provides a comprehensive summary of the code review.

2.7.2 A Tutoring System to Learn Code Refactoring (Keuning *et al.*, 2021)

This paper describes a tutoring system, Refactor Tutor, designed to address the looming problem in which tutoring systems and assessment tools have failed to provide solutions for. This problem introduced by the paper is the lack of code quality integration into these systems and tools for students or beginner programmers. Code quality is important for professional software development as bad code is hard to understand, maintain and test (Keuning *et al.*, 2021).

Refactor Tutor provides students with exercises that begin with functionally correct code but is poorly written. Students are tasked with improving this code by refactoring the code in a step by step manner. The system offers hints and feedback to guide students throughout this process. These hints are presented in a hierarchical structure, the default hint at the top, allowing students to click on explain more button to get a more detailed hint.

One of the biggest benefits of the system is that it was created for beginners in programming. It stresses the relevance of using simple words that are not vague and ambiguous. Again, there is a sequence to follow while learning using this system. This means that apart from just providing directions which get more detailed progressively, it guides students depending on their own demands. The system also focuses on issues that are commonly faced by novice programmers. These issues are identified through research and through the input of experienced teachers. The feedback and hints provided by the system are based on

the suggestions from these teachers. The figure below is a screenshot of the web application for the tutoring system.

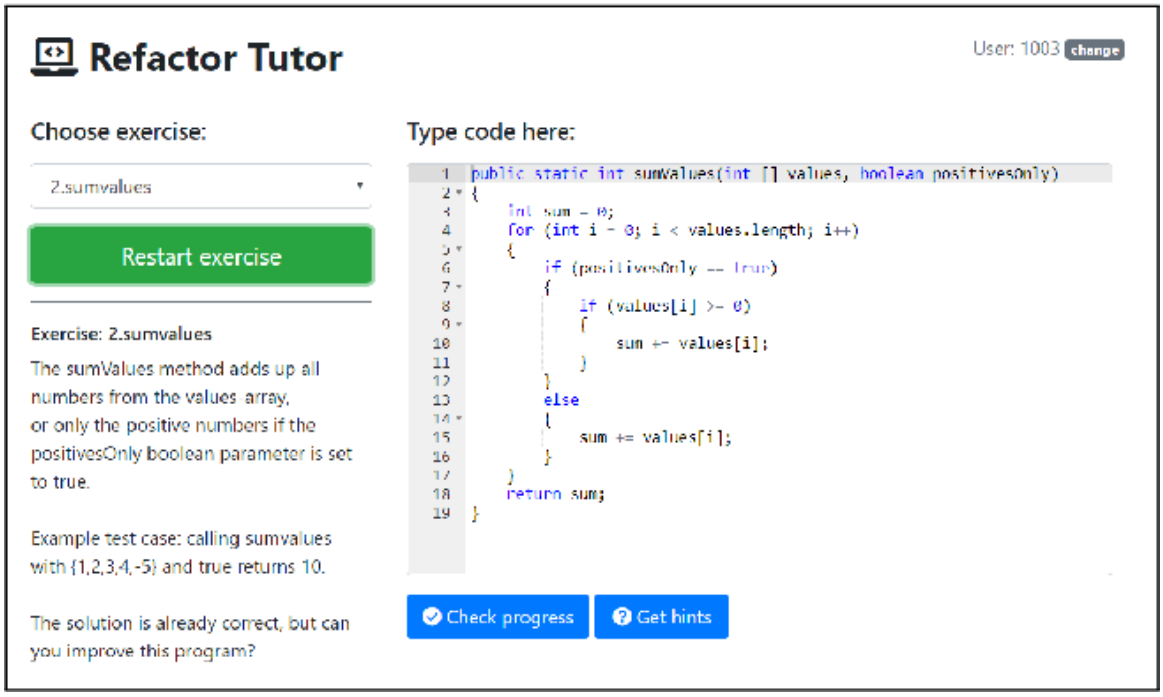

Figure 2.11: Web application for the tutoring system. [Source: (Keuning *et al.*, 2021)]

An evaluation of the system was conducted by comparing the system's feedback with suggestions from teachers on how to improve student code. The results showed a strong correlation between the suggestions offered by the system and those provided by the teachers. Additionally, a preliminary study with students indicated that the system's hints help solve refactoring exercises.

A. Limitations of Refactor Tutor and How the Proposed Tool Addresses Them

The tutoring system's effectiveness relies on a predefined set of exercises and rules, which may not cover the large potential scenarios encountered in real world coding. Additionally, the system requires manual effort from teachers to create model solutions for generating new rules. Static analysis and LLMs have several advantages over the tutoring system. First, LLM based systems can learn and adapt from massive codebases, find more issues and keep up with changing coding practices. No need for a predefined set of exercises and manual rule creation by teachers in the tutoring system. LLMs can also analyse code for security issues, performance issues and even consider the code's context for more precise feedback.

Table 2.2: Comparison of features between the Refactor Tutor and the Proposed Intelligent Code Review System

Feature	Refactor Tutor	Proposed Intelligent Code Review Tool
Methodology	Predefined set of exercises and pre defined rules.	Machine learning on vast code repositories provided by the pretrained LLM.
Adaptability	Limited, requires manual effort for expansion.	Scalable, learns and adapts automatically.
Feedback Style	Step by step guidance for refactoring each coding violation exercise.	Nuanced feedback based on code context.
Knowledge Base	Static set of teacher defined rules.	Continuously learns from code repositories.
Efficiency	Requires teacher effort to create model code violation exercises.	Does not require external effort to detect code violations.

2.7.3 Codacy

There are many SCA tools out there, but Codacy is a user friendly and all in one solution to make code review easier. One of Codacy's biggest strengths is the ease of use. The platform has a user friendly interface that integrates with popular development workflows and version control systems. So, developers can just add SCA to their existing workflow and keep improving their code. Codacy also supports many programming languages, making it a great tool for teams working on many projects.

Although user-friendliness is a nice feature, Codacy has more to offer than the basics. The platform can find many code quality issues because it uses a powerful set of static analysis engines. This includes finding security holes, following coding standards, and finding ways to refactor existing code. Codacy also gives developers concise and useful feedback to fix issues and improve the quality of their code.

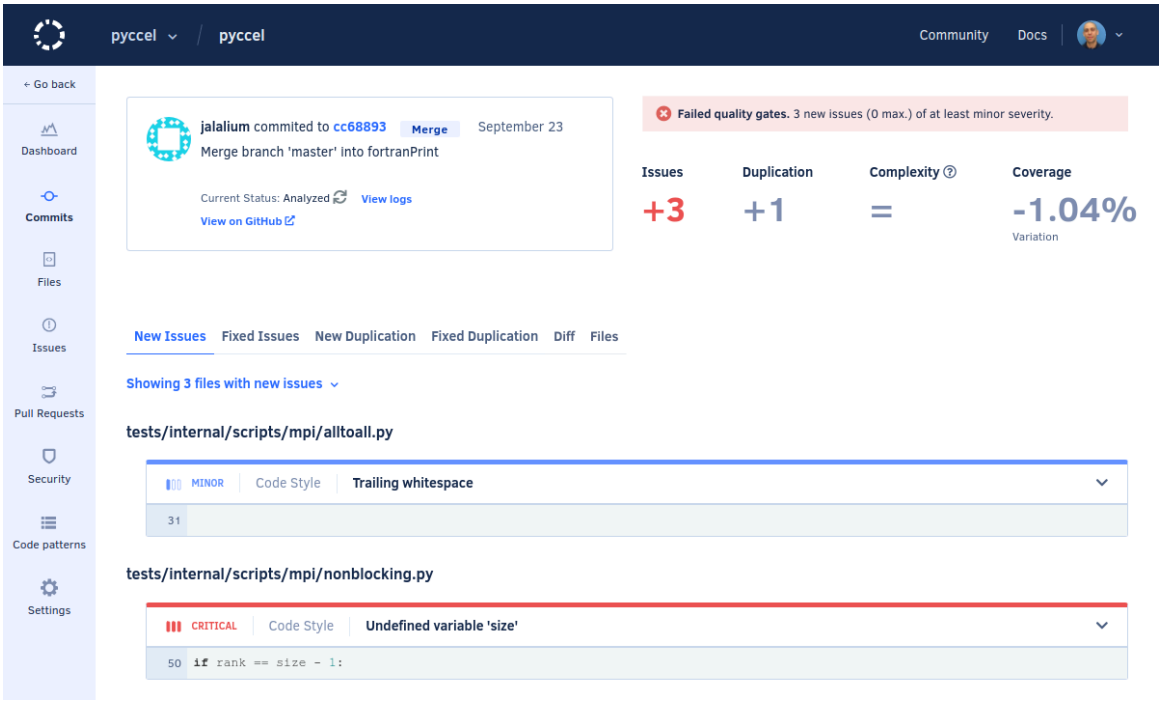

Figure 2.12: Overview of issues in recent commits on Codacy Dashboard

A. Limitations of Codacy and How the Proposed Tool Addresses Them

Codacy offers a user friendly and comprehensive SCA solution, but it has limitations compared to the proposed intelligent code review tool. Codacy is a great tool for professional code quality management but as it does not offer a nuanced explanations or feedback based on the style issues found, it is not completely helpful to a beginner in any programming language trying to learn how to adhere to programming conventions or coding styles of that language. The intelligent code review tool, however, provides contextual explanations of the code violations found in the user submitted code. Ensuring the user understands and learns proper programming styles.

Table 2.3: Comparison of features between the Refactor Tutor and the Proposed Intelligent Code Review System

Feature	Codacy	Proposed Intelligent Code Review Tool
Methodology	Predefined set of exercises and pre defined rules.	Machine learning on vast code repositories provided by the pretrained LLM.
Adaptability	Limited, based on rule updates.	Scalable, learns and adapts automatically.
Feedback Style	Simple error and warning messages.	Nuanced feedback based on code context.

Feature	Codacy	Proposed Intelligent Code Review Tool
Scope of Issues Detected	Defined set of coding.	Broader range of issues and best practices

2.8 Summary of Literature Review

This literature review looked at different methods and approaches to improving code quality. From Modern Code Review tools to automated Code Review tools, a code review system utilizing static analysis and LLMs (Large Language Models) emerged as a more promising solution LLM's can learn continuously so can identify more code quality issues and adapt to changing coding practices. Additionally, an LLM can also analyse code for context specific nuances, introducing more comprehensive feedback than a static set of rules.

In general, LLM based code review is a more scalable, flexible, and efficient way to enhance code quality—especially when it comes to getting students ready for the demands of the software development industry.

CHAPTER THREE
METHODOLOGY OR SYSTEM ANALYSIS AND DESIGN

3.1 Preamble

This section details the analysis and design of the proposed system. It presents the requirements for the system, the techniques taken to achieve the system as well as diagrams to model the proposed system.

3.2 The Proposed System

This proposed project is a web based application designed to streamline code review. Users can connect their codebases through repositories like GitHub or import their code directly into the application. Once the code is imported, the application conducts an extensive review, highlighting areas for improvement and offering recommendations. Additionally, the system provides explanations and justifications for these suggestions, utilizing intelligent responses from a language model.

3.3 Requirement Analysis

In software development, requirement analysis includes the descriptions of the services that would be provided by the system, as well as its operational limits. It lays down the

features that are required to suit the users' requirements, these requirements are then further separated into functional requirements, non functional requirements and user experience requirements.

3.3.1 Functional Requirements

Functional requirements of the system are specific functionalities or services that the system is expected to perform. They describe what the application should do and include operations, activities, computational tasks, data manipulation, user interface behaviour, and more. These requirements are as follows:

- **(i)** Users should be able to input their code or to link their respective codebases from code hosting sites.
- **(ii)** The system should analyse the code and provide feedback on areas that need improvement.
- **(iii)** The system should generate appropriate and contextually relevant explanations and justifications based on the content of the code and the suggestions made.
- **(iv)** The system should allow users to view the code and the suggestions side by side.
- **(v)** The system should allow users to accept or reject the suggestions made.
- **(vi)** The system should provide a summary of the code review.
- **(vii)** The system should allow users to download the reviewed code

3.3.2 Non-Functional requirements

Non functional requirements are not directly related to the system functionality but rather are defined in terms of system performance. They explain the system behaviour, features and overall attributes that can affect the user's experience. The non functional requirements for the system in this study are as follows:

- **(i)** The system should be easy to use and navigate.
- **(ii)** The system should provide clear and concise feedback.
- **(iii)** The system should provide explanations and justifications that are easy to understand.
- **(iv)** The system should provide a summary of the code review.
- **(v)** The system should allow users to download the reviewed code.

3.3.3 User Experience Requirements

User experience requirements are the factors that specify the expected things the user should encounter while making use of the system and the outcome of their experience. In the case of this system, the user experience requirements include:

- **(i)** The system should be easy to use and navigate.
- **(ii)** The system should provide clear and concise feedback.
- **(iii)** The system should provide explanations and justifications that are easy to understand.
- **(iv)** The system should provide a summary of the code review.
- **(v)** The system should allow users to download the reviewed code.

3.4 Physical Design

A physical design represents the physical elements of a software system and relates to the actual input/output operations of the system. The focus is on how data is entered into the system, validated, processed, and output. It has two major categories, the input design and the output design. For this system, the physical design is concerned with how the code is inputted into the system, how the system processes the code, and how the system outputs the results of the code review. These design considerations are essential to generate a working system that specifies all the features of a candidate system.

3.4.1 The Proposed System Architecture

The system architecture describes the structure of the system and the components that make up the system. It provides a high level overview of how the system will be designed and how the components will interact with each other. The system architecture for the proposed system is detailed in this section.

The proposed intelligent code review application is composed of several components designed to enhance the code review process. These components include the Code Importer or Fetcher mechanism (Github integration or manually importing code), a simple static analysis engine, a user friendly interface for users, a recommendations module or generator (CodeLlama Model), and a comprehensive reporting system. The functions of the components of the architecture are as follows:

A. Code Importer/Fetcher

This component allows users to import their code into the system either by linking their codebases from code hosting repositories like GitHub or by manually uploading their code. This component serves as the entry point for the code review process.

B. Static Analyzer

The system's static analyzer plays a crucial role in examining user submitted code. It begins by tokenizing the code, parsing it, and constructing an abstract syntax tree (AST) to comprehend the code's structure and semantics. Through this analysis, it identifies code quality concerns, adherence to coding conventions, and areas for enhancement. The static analyzer encompasses several key components:

(i) Lexer: The lexer tokenizes raw source code into tokens, facilitating further analysis.

(ii) Parser: The parser constructs an Abstract Syntax Tree (AST) using a recursive descent parsing method, capturing code hierarchy systematically.

(iii) Semantic Analyzer: The semantic analyzer verifies the code's semantic correctness, ensuring adherence to programming language rules and identifying logical errors.'

(iv) Code Rule Enforcement: This component checks code against coding standards, detecting stylistic issues and ensuring consistency.

(v) Report Generation: Detailed reports are created, highlighting the identified issues, the AST, and code snippets. This report serves as a query that would be fed to the LLM for contextual recommendations or suggestions based on the violations found in the analysis.

C. User Interface

The user interface component provides an intuitive and user friendly platform for users to interact with the system. It offers functionalities for code submission, viewing analysis results, accepting or rejecting suggestions, and downloading the reviewed code.

D. Recommendations Module

The recommendations module is powered by the CodeLlama 7B Instruct GGUF Model and it generates contextually relevant suggestions based on the code analysis results. The model's deep learning capabilities allows the module to provide nuanced feedback, explanations, and justifications for the identified issues.

E. Reporting System

The reporting system compiles the analysis results, recommendations, and justifications into a comprehensive report. This report summarizes the code review process, highlighting the

identified issues, suggested improvements, and contextual explanations. Users are allowed to accept the suggested changes from this report.

3.5 Logical Design

The logical design of a system describes the functional requirements of the system and how these requirements are implemented. It refers to the conceptual and abstract representation of a system's architecture, functionality, and components. For the proposed system, the goal of logical design is to define the system's structure, data flow, interfaces, and interactions between various components in a way that is independent of any particular technology or platform. The logical design of the proposed system is detailed in this section.

3.5.1 Use Case Diagram

A use case diagram is a visual representation of the interactions between users and a system. It illustrates the various ways users can interact with the system and the system's responses to these interactions. The use case diagram for the proposed system shows that a user can interact with the system by importing their code, analyzing the code, viewing the analysis results, accepting or rejecting suggestions, and downloading the reviewed code. The system, in turn, processes the code, generates recommendations, provides explanations, and compiles a summary report for the user. The use case diagram provides a high level overview of the system's functionality and user interactions, outlining the key features and capabilities of the proposed system, the diagram is shown in the figure below.

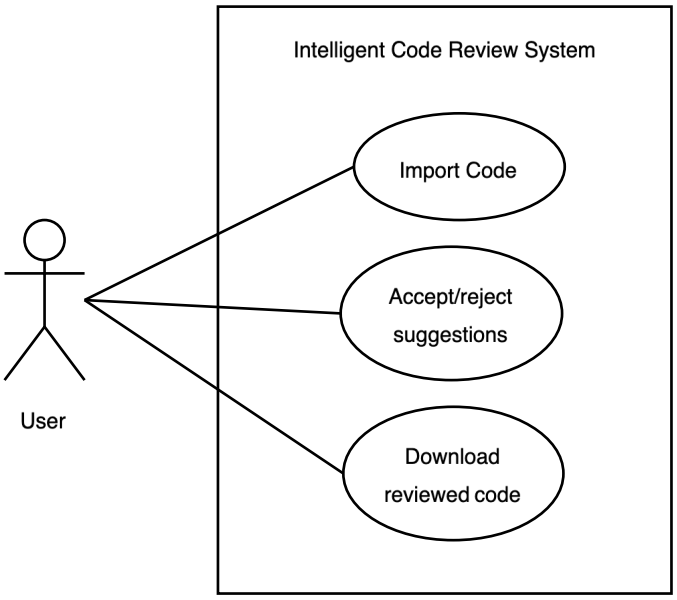

Figure 3.1: Use case diagram for the proposed system

3.5.2 Sequence Diagram

A sequence diagram shows how objects interact in a particular scenario of a use 36 case. It illustrates the sequence of messages exchanged between objects, highlighting the order of interactions and the flow of control in a system. The sequence diagram of the system is shown in the figure below,

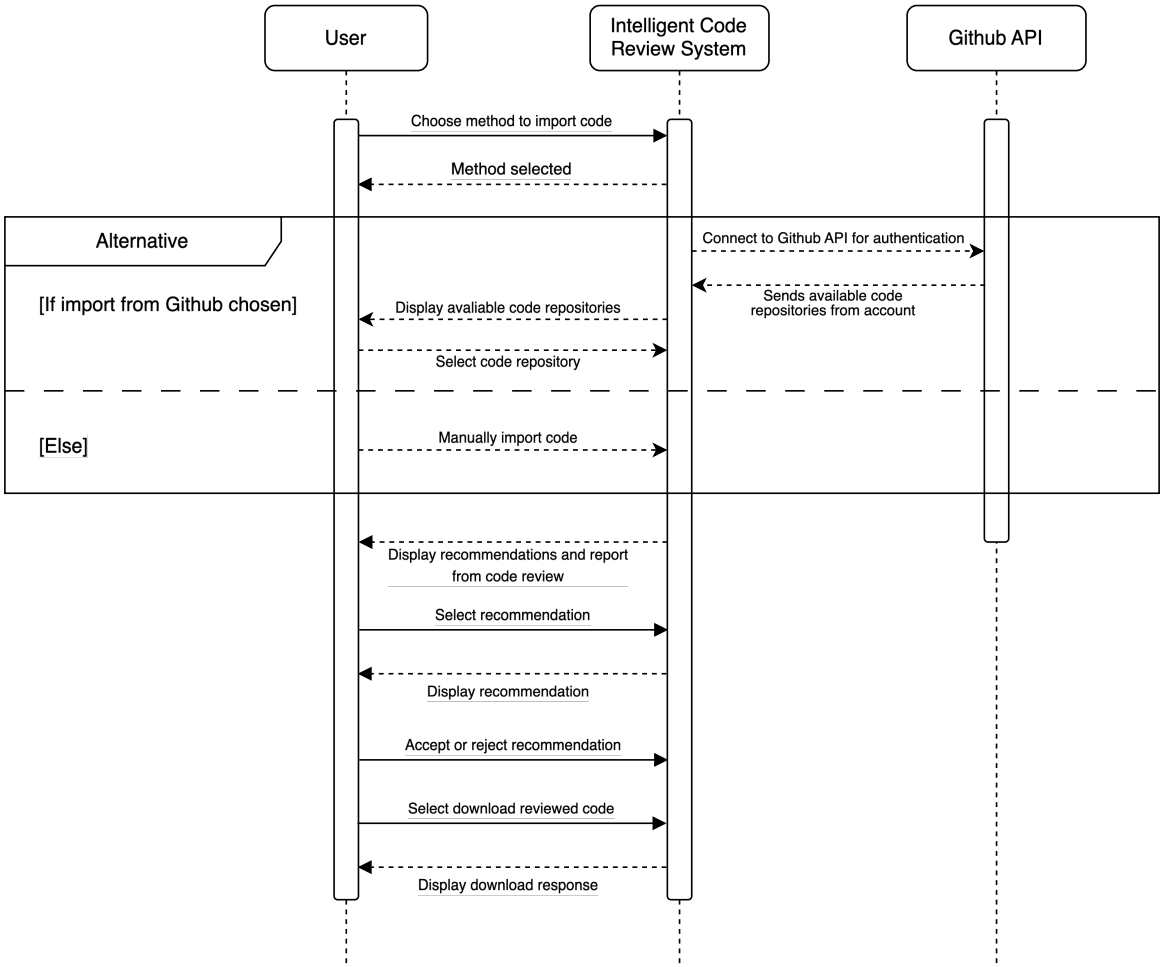

Figure 3.2: Sequence diagram for the proposed system

3.6 Conceptual Design

The conceptual design of the system outlines the high level structure and components of the system. This section shows the Entity Relationship Diagram (ERD) for the proposed system, detailing the entities and their relationships. The conceptual diagram is shown in the figure below.

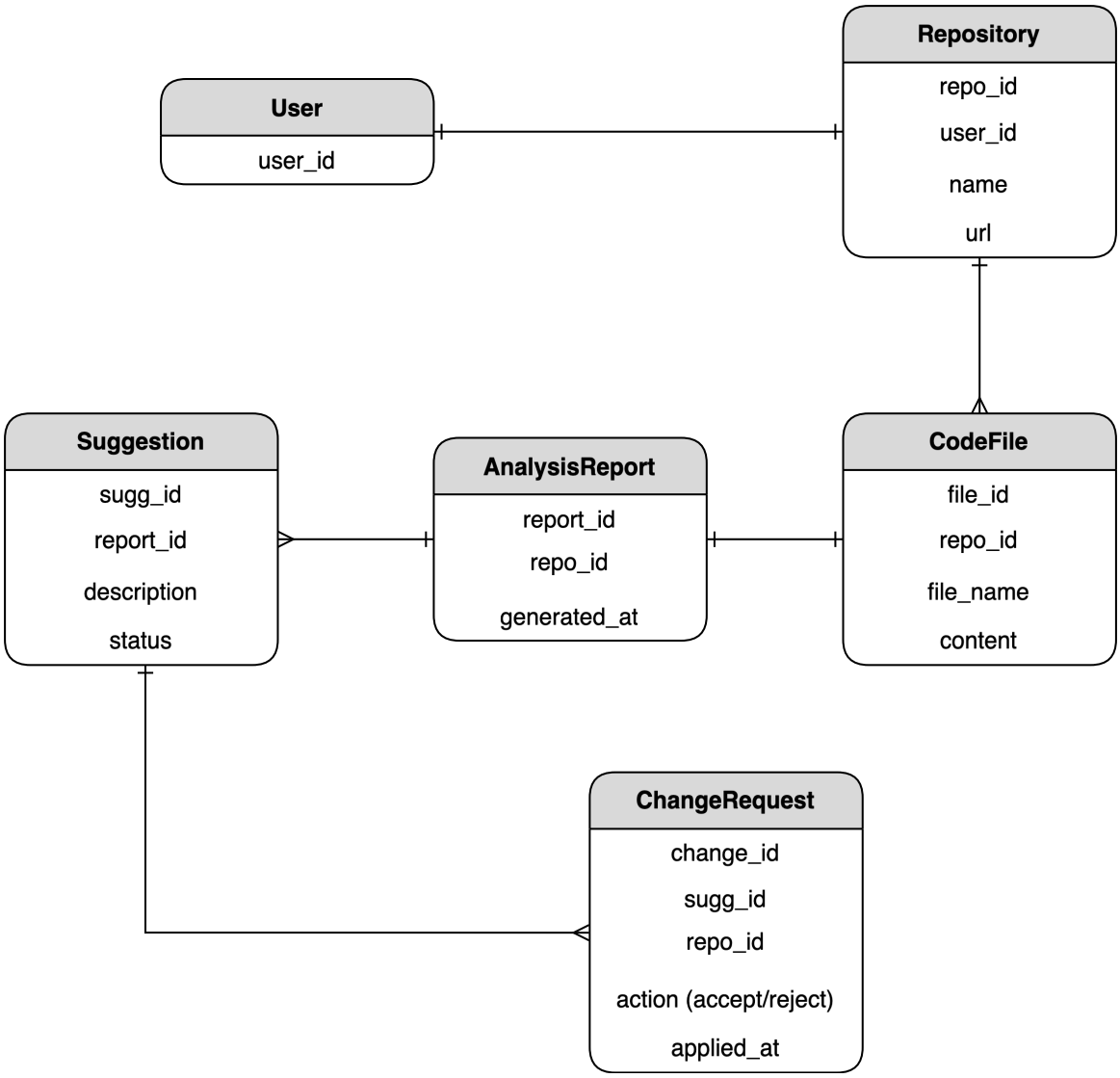

Figure 3.3: Entity relationship diagram for the proposed system

CHAPTER FOUR
SYSTEM IMPLEMENTATION AND EVALUATION

4.1 Preamble

This chapter of the report entails the actual implementation of the proposed intelligent code review system. It details the tools used for the implementation, the hardware and software requirements, the development methodology, and the evaluation of the system.

4.2 System Requirements

System requirements are the specifications that define the functionalities and features of the system. They detail the hardware and software requirements necessary for the system to operate effectively. It includes the software and hardware requirements for the system.

4.2.1 Hardware Requirements

These are the requirements necessary for the hardware components to run the system. The hardware requirements for the proposed system are shown in the table below:

Table 4.1: Hardware Requirements

S/N	Requirement	Hardware
1	Processor	Intel Core i7 / Apple M1 Chip or higher
2	Graphics Processing Unit (GPU)	Nvidia GTX 1660 or 2060, AMD 5700 XT, or Nvidia RTX 3050 or 3060
3	Memory (RAM)	16GB Ram or Higher
4	Architecture	64 bit (x64) architecture / ARM x64
5	Secondary Storage	32GB SSD or higher

4.2.2 Software Requirements

These are the requirements needed by the programs of the system to run successfully. They must be pre installed and set for the proposed system to run smoothly. The software requirements for the proposed system are shown in the table below.

Table 4.2: Software Requirements

S/N	Requirement	Software
1	Operating System	Windows 10, macOS Big Sur or higher, Ubuntu 20.04 or higher.
2	Integrated Development Environment (IDE)	Visual Studio Code, PyCharm, IntelliJ IDEA, or any other IDE.

S/N	Requirement	Software
3	Programming Language	Python 3.8, Javascript (NodeJs), Rust.
4	Supported Browser	Google Chrome, Safari, Mozilla Firefox

4.3 System Implementation

The system implementation of the intelligent code review tool is detailed in this section.

4.3.1 Setup and Configuration

The setup of the development environment entails various installations and configurations for the frontend of the tool and backend of the tool:

A. Frontend and Backend Configuration

In the frontend of the tool, React was used in the development of the user interface. First, the React application is initialised using the Vite bundler, which provides a fast development environment for React projects. Other dependencies, such as Tailwind CSS for the styling or React Redux Toolkit and Query for state management and API consumption, were installed using the Node.js package manager for JavaScript.

The backend of the tool is implemented using Express.js for the server side, where all the API endpoints are created and served. Node.js is also used to install all the dependencies of the backend server such as jsonwebtoken that is used in the generation of JWT tokens for all GitHub API endpoints. For the integration with the LLM, python was used, the Flask python framework to run a special server for the LLM, Llama-cpp that is used to load and interact with the GGUF version of CodeLlama.

B. Integration and Testing Phase

During the development of the application, the integration phase involved several key modules and components to ensure it fulfilled all the functional requirements of the application. For the import module of the application, integration with GitHub was essential to enable users have access to their repositories if need be. GitHub has several criteria needed to integrate with external applications. The first was to create a GitHub App to get a client id and client secret which are both needed in installing the app on a GitHub users account. After the creation of the GitHub App, generating Access Tokens would be required to consume any of the GitHub API's such as the get all users repo endpoint or get content from a specific repository's endpoint. All API consumption on the backend was done using JavaScript's Fetch function.

Another aspect of the import module was the manual import, which allows a user to import a single JavaScript file from their local machine. The code is then sent to the backend using the RTK Query API slices created for that purpose. Before integration with the backend of the application, CodeLlama was tested using a small dataset of code snippets with their violations. This testing was done to determine the efficiency of the LLM in explaining code violations passed to it. The editor module or component was integrated using a simple Syntax Highlighter Node package. This enabled all code snippets to be highlighted according to the JavaScript language. The code snippets and suggestions were passed from the backend after a user imports the code and a review on the code is done.

Finally, in testing all components of the application, each process of connecting a GitHub account to reviewing and downloading the reviewed code was tested several times. Each bug encountered was analysed and fixed to ensure that the application worked as intended.

4.4 Development Methodology

The software development methodology applied to implement the developed system is an agile approach. Agile methods are mainly iterative and incremental by nature and share key features of collaboration, flexibility, and adaptability. The agile approach always involves continuous feedback, a rapid development cycle, and responses to changing needs. The process is broken down into sprints, each orienting toward certain features or functionalities. The agile methodology allows the development team to offer a working product incrementally, thereby ensuring that the user requirements and expectations for the system are met. The study involved initially following the waterfall model to define the system requirements, analyse the system, design the system, and implement the system. However, the development process was later shifted to an agile approach to accommodate changing requirements, incorporate feedback, and deliver a working system incrementally.

4.5 Implementation Tools

These are the tools used to implement the system. They include the programming languages, libraries, and frameworks used to develop the system.

4.5.1 JavaScript

JavaScript is a versatile programming language that is widely used for web development. It is essential for building interactive web applications and dynamic content on websites. In the proposed system, JavaScript was used for the front end development, utilising the React

Framework with the vite bundler, and was also used in part of the backend development for serving endpoints and other API logic.

4.5.2 Python

Python is a high level programming language known for its simplicity and readability. It is widely used for various applications, including web development, data analysis, and artificial intelligence. In the proposed system, Python was used for the backend development, to load and prompt the CodeLlama model. Libraries such as the Torch, Transformers (from hugging face) and more was utilized in the preparation of the model for the software system.

4.5.3 Rust

Rust is a systems programming language known for its performance, reliability, and memory safety. It is often used for low level programming tasks and building high performance applications. In the proposed system, Rust was used for the static analysis engine, which tokenizes, parses, and analyses the user submitted code. The Rust programming language was chosen for its speed, safety features, and suitability for building efficient code analysis tools.

4.5.4 Visual Studio Code

Visual Studio Code is a popular code editor developed by Microsoft. It is widely used by developers for its versatility, extensibility, and rich feature set. In the proposed system, Visual Studio Code was used as the primary code editor for developing the system components.

4.6 Program Modules and interfaces

This section details the sections of the user interface and modules of the intelligent code review system.

4.6.1 The Import Module

This is the screen where users choose where to import their code from. They can either import from GitHub or upload their code directly. These screens show the integration with Github to retrieve user repositories. In figure 4.2, it alerts the user to authorize the application before installing it on their GitHub account. One the authorize button is clicked it take the user to the GitHub portal where the application can then be installed. The import module screens are shown in figures 4.1 to 4.3.

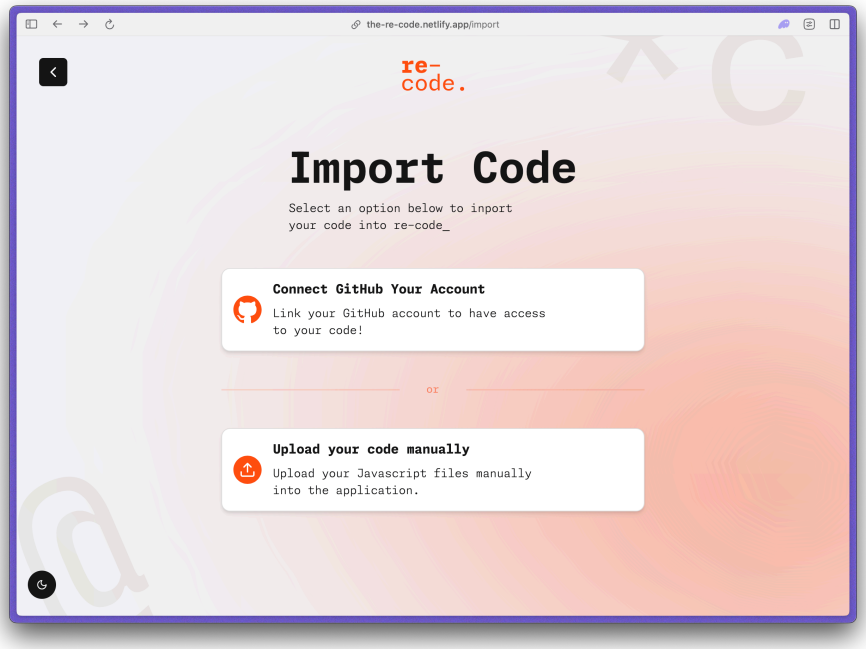

Figure 4.1: The import page

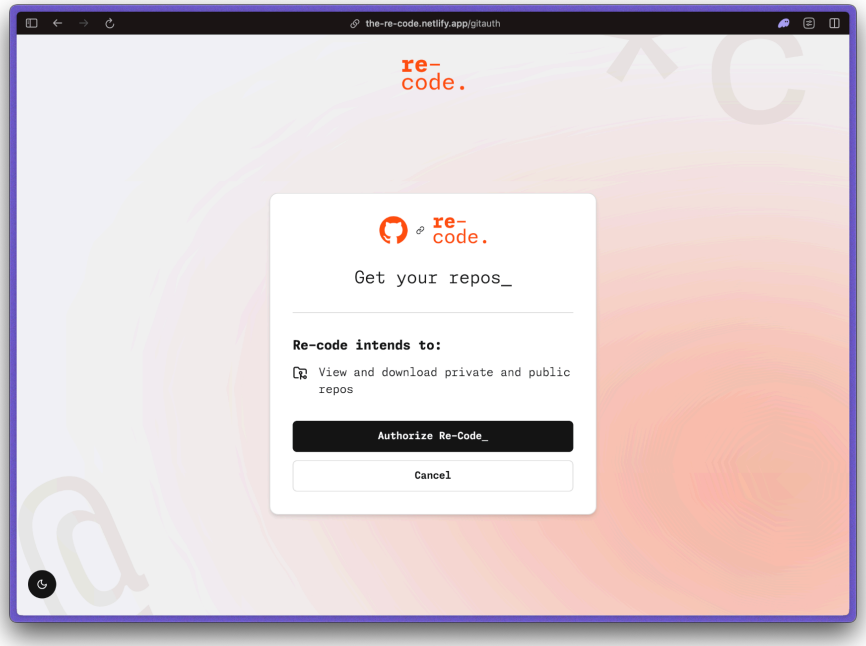

Figure 4.2: The authorization page for GitHub Authentication.

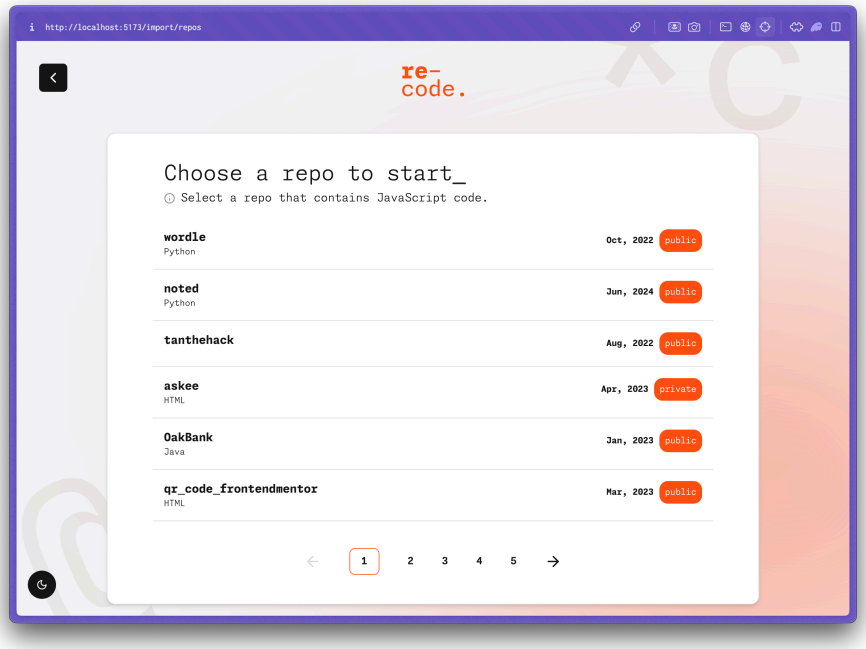

Figure 4.3: The import page for choosing a specific repository

4.6.2 The Code Review Module

This module is where the after the code has been reviewed, the suggestions are presented to the user in a simple read-only editor layout. In the figure below, it shows the error lines in a light orange colour to indicate to the user where the code violations were found. A user can then click on the error lines to view the suggestions from the code review. The suggestions can be expanded as show in figure 4.5, to provide more explanations and resources based on the rule violated. The screens are shown in the figures 4.4 to 4.5.

Figure 4.4: The code review page

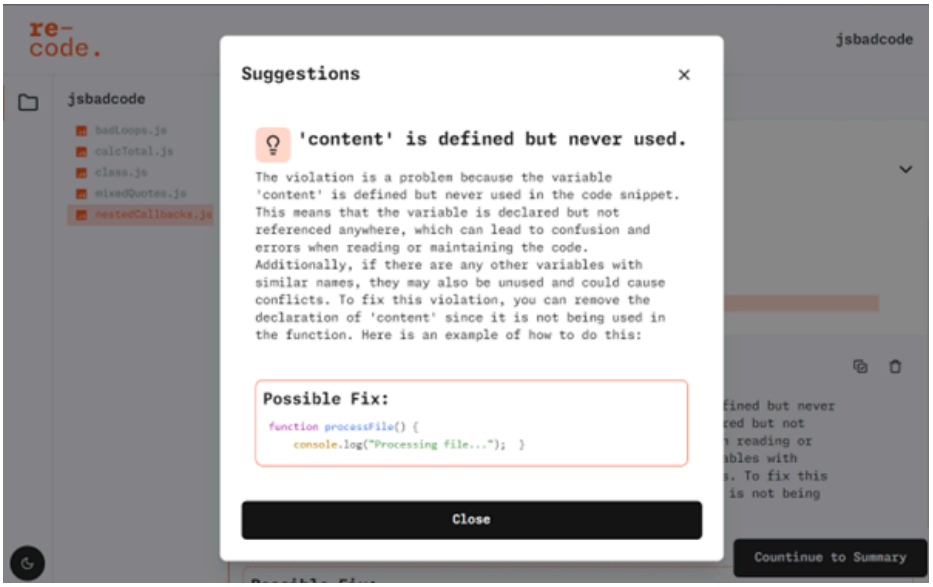

Figure 4.5: The suggestions popup

4.6.3 The Summary Module

After the code review process, the user can continue to the summary page where a simple breakdown of the review will be presented to the user. The user can then proceed to download the reviewed code. The module is show in figure 4.6 below.

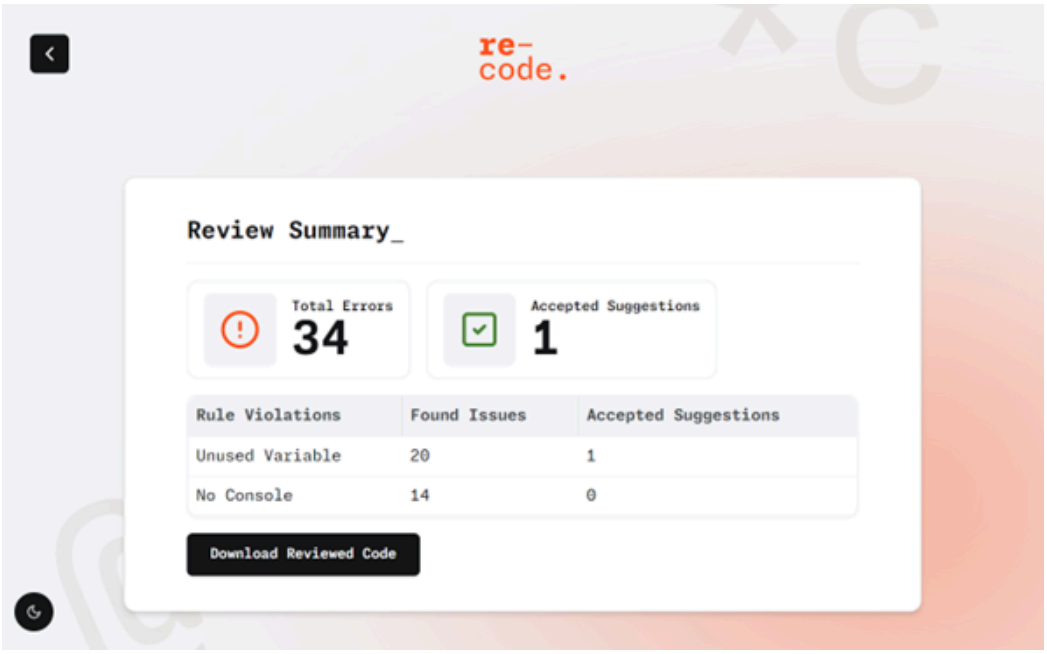

Figure 4.6: The summary screen

4.7 Evaluation of the System

The evaluation of the intelligent code review system was conducted through questionnares with users who interacted with the system. The evaluation technique used in the study was

4.7.1 Interview Results

The evaluation of the intelligent code review system involved interviews with users who interacted with the system. The interviews provided feedback on the system's performance, usability, and user satisfaction. The results of the interviews are summarized below.

A. User Experience

(i) Initial Experience Rating: Most users rated their initial experience with the code review system positively, with ratings predominantly falling between 2 and 4 on a 5-point scale.

(ii) Interface Intuitiveness Rating: Most users found the interface to be intuitive, with several describing it as "Intuitive" and some noting it as "Very Intuitive". This suggests that the design and layout of the user interface were well-received.

B. Helpness and Clarity of Feedback

(i) Helpfulness of Feedback: The feedback from the system was generally considered helpful, with ratings mostly in the range of 1 to 4, indicating a varied but generally positive reception.

(ii) Clarity of Feedback: The clarity of the feedback was consistently rated as "Clear" or "Very Clear", demonstrating that the explanations and suggestions provided by the system were easily understood by the users.

C. Comparison to Human Reviewers and Other Tools

(i) Feedback Comparison: Users compared the feedback from the system to that received from human reviewers. Responses varied, with some users indicating the system's feedback was "About the Same" as human feedback, while others found it "Better".

(ii) Comparison to Other Tools: When comparing the intelligent code review system to other code review tools, most users rated it as "Better" or "Much Better", suggesting that the system offers a competitive or superior alternative to existing tools.

D. Overall Satisfaction

(i) Satisfaction Rating: Overall satisfaction with the system was moderate, with ratings primarily around 2 to 3 on a 5-point scale. This indicates room for improvement but a generally acceptable level of satisfaction among users.

(ii) Specific Likes and Dislikes: Users appreciated the user interface (UI) and the clarity of the feedback. Comments such as "the UI", "The way it is explained", and "The feedback was easy to understand and explain" highlight these positive aspects. Common criticisms included system lag and occasional crashes, with users noting "The lags and occasional crashes" and "the system lag". Some users also mentioned the slowness of loading as a drawback.

4.7.2 Comparison of Current System to Reviewed Systems

One of the primary distinctions between the intelligent code-review tool and the FrenchPress plugin lies in the target audience. While FrenchPress caters specifically to intermediate Java developers, the tool is designed to provide personalized feedback for a broad spectrum of skill levels, ranging from beginners to experienced programmers. This inclusivity ensures that a wider range of users can benefit from the tool's capabilities, fostering a more comprehensive learning environment. The adaptive nature of the intelligent code-review tool makes it suitable for varied educational contexts, enhancing its overall utility compared to the more narrowly focused FrenchPress.

Another significant limitation of FrenchPress is its fixed list of code issues. The plugin relies on a predefined set of style issues and violations, limiting its scope for comprehensive code quality assessment. In contrast, the intelligent code-review tool leverages an extensive array of coding standards from established guidelines, providing a more thorough analysis. This flexibility allows the tool to address a broader range of coding practices and standards, ensuring more robust code quality assessment. By utilizing a dynamic set of code violations, the intelligent code-review tool surpasses FrenchPress in its ability to adapt to evolving coding conventions and practices.

The use of a Language Model (LLM) for feedback and explanations is another area where the tool excels. Unlike FrenchPress, which offers static, predefined feedback tailored to specific violations, the intelligent code-review tool generates scalable and dynamic feedback. This adaptability enables the tool to provide nuanced and contextually relevant explanations, fostering a deeper understanding of coding standards among users. This feature is particularly beneficial for beginners, as it helps them grasp the rationale behind coding conventions, enhancing their learning experience. The integration of LLM technology ensures that the tool remains relevant and effective in providing up-to-date and comprehensive feedback, a significant advantage over the static feedback mechanism of FrenchPress.

4.7.3 Discussion

The evaluation of our intelligent code-review tool reveals several strengths and areas for improvement, providing a comprehensive view of its performance and user satisfaction. One of the notable strengths is the user interface's intuitiveness, which 85.7% of users found easy to navigate. Apart from this, the feedback provided by the tool was generally perceived as clear and helpful, with users appreciating the detailed and understandable explanations.

However, some areas require attention as users indicated problems, such as system lag and occasional crashes, which were common criticisms. Addressing these issues could significantly enhance user satisfaction and system reliability. Comparatively, the intelligent code-review tool offers distinct advantages over the FrenchPress plugin and other reviewed systems. Unlike FrenchPress, which targets intermediate Java developers, our tool is designed to cater to a wide range of skill levels, from beginners to advanced programmers. This inclusivity makes the tool more versatile and beneficial across different educational contexts.

Furthermore, the intelligent code-review tool's dynamic approach to identifying code violations, drawing from an extensive array of coding standards, ensures a more comprehensive assessment. This flexibility allows the tool to adapt to evolving coding practices, providing users with up-to-date and relevant feedback. An advantage it has over the reviewed systems such as FrenchPress and Refactor Tutor. Another significant advantage of the intelligent code-review tool is its use of a Language Model (LLM) for generating feedback and explanations. This approach offers scalable and dynamic feedback, which is more nuanced and contextually relevant compared to the static, predefined feedback of FrenchPress. The LLM's ability to provide detailed justifications for coding suggestions enhances users' understanding of coding conventions and practices.

CHAPTER FIVE
CONCLUSION AND RECOMMENDATIONS

5.1 Summary

An intelligent code review tool is a system that enables a user to receive dynamic suggestions to code violations found within the review process. This is tool is vital in the improvement of code quality and adherence to programming conventions in the software development industry. The use of this system in the early stages of learning to code can not only improve developers code quality but also improve a programmer's code

comprehension. A pretrained Large Language Model was used in this project to provide contextual explanations on the code violations fond during the review process. Static Analysis techniques were also utilised in the project to identify code violations in user submitted code.

During the evaluation, it was observed that the integration of the LLM and static analysis techniques offered a comprehensive approach to code review. The intelligent code review tool was tested in various scenarios, demonstrating its effectiveness in improving code quality and user understanding, with 71.5% of interview responses indicating the system's suggestions quality better than human reviewers and 71.4% of user's indication the clarity of the feedback from the system. The results highlighted that the system's dynamic feedback mechanism and extensive code violation coverage set it apart from traditional tools like the FrenchPress plugin. By addressing a broader range of coding issues and providing adaptive, contextually relevant feedback, the intelligent code review tool enhances the learning experience for programmers at all skill levels, ultimately contributing to better software development outcomes.

5.2 Recommendations

The following are recommendations for the intelligent code review tool:

- **(i)** To address the varied reception of feedback clarity, the system will benefit from implementing fine-tuning the LLM for more personalized feedback.
- **(ii)** The system's performance can be improved by optimizing the backend process. Either by hosting the LLM on better infrastructure or by using a quantized version of the model. This may cause quality loss but can be considered as a trade-off for performance.
- **(iii)** As a learning tool, introduction of features such as learning resources, or a collaborative environment can allow users to enhancing their learning process.

5.3 Conclusion

The development and implementation of the intelligent code review tool marks a significant advancement in enhancing code quality and adherence to programming conventions within the software development industry. Leveraging state of the art technologies such as a pretrained Large Language Model (LLM) and robust static analysis techniques, this project has demonstrated its capability to provide dynamic and contextual suggestions for code violations, thereby supporting developers at various skill levels in improving their coding practices. Throughout the project, extensive research and understanding of the LLM

to deliver precise and informative feedback on code quality. By surpassing traditional approaches with its adaptive feedback mechanisms, the tool not only identifies a wide range of coding issues but also educates users on best practices, fostering a deeper understanding of programming standards.

Looking forward, the tool's practical applications have streamlined code review processes, enabling real-time analysis and actionable insights into codebases. The implementation of an intuitive user interface and responsive feedback system has garnered positive feedback from users, highlighting its effectiveness in enhancing developer productivity and code maintainability. As digital environments continue to evolve, the intelligent code review tool stands poised to drive further innovations in software development practices. By addressing current limitations and anticipating future challenges, this project paves the way for continued advancements in automated code analysis and education, contributing to a more efficient and quality-driven software development ecosystem.

REFERENCES

Alberto, B., & Christian, B. (2013). Expectations, outcomes, and challenges of modern code review. *2013 35th International Conference on Software Engineering (ICSE)*, 712–721. https://api.semanticscholar.org/CorpusID:220663293

Albuquerque, D., Guimarães, E., Perkusich, M., Almeida, H., & Perkusich, A. (2023). Integrating Interactive Detection of Code Smells into Scrum: Feasibility, Benefits, and Challenges. *Applied Sciences*, *13*(15). https://doi.org/10.3390/app13158770

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1), 53–54. https://doi.org/10.1186/s40537-021-00444-8

Alzubi, J., Nayyar, A., & Kumar, A. (2018). Machine Learning from Theory to Algorithms: An Overview. *Journal of Physics: Conference Series*, *1142*, 12012–12013. https://doi.org/10.1088/1742-6596/1142/1/012012

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., & Systa, K. (2016). The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, 9–16. https://doi.org/10.1109/MTD.2016.8

Badampudi, D., Unterkalmsteiner, M., & Britto, R. (2023). Modern Code Reviews—Survey of Literature and Practice. *ACM Trans. Softw. Eng. Methodol.*, *32*(4). https://doi.org/10.1145/3585004

Bartneck, C., Lütge, C., Wagner, A., & Welsh, S. (2021). What Is AI?. In *An Introduction to Ethics in Robotics and AI* (pp. 5–16). Springer International Publishing. https://doi.org/10.1007/978-3-030-51110-4_2

Bhat, S., Bhirud, R., & Bhokare, V. (2022). Survey on Various Syntax Analyzer Tools. *International Journal for Research in Applied Science and Engineering Technology*, *10*, 109–115. https://doi.org/10.22214/ijraset.2022.46588

Bielik, P., Raychev, V., & Vechev, M. (2017). *Learning a Static Analyzer from Data*. 233–253. https://doi.org/10.1007/978-3-319-63387-9_12

Blau, H., & Moss, J. E. B. (2015). FrenchPress Gives Students Automated Feedback on Java Program Flaws. *Proceedings of the 2015 ACM Conference on Innovation and*

Technology in Computer Science Education, 15–20. https://doi.org/10.1145/2729094.2742622

Carvalho, L. P. d. S., Novais, R. L., Salvador, L. d. N., & Mendonça, M. G. (2017). An Ontology-based Approach to Analyzing the Occurrence of Code Smells in Software. *International Conference on Enterprise Information Systems*. https://api.semanticscholar.org/CorpusID:21143193

Chirvase, A., Ruse, L., Muraru, M., Mocanu, M., & Ciobanu, V. (2021). Clean Code - Delivering A Lightweight Course. *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, 420–423. https://doi.org/10.1109/CSCS52396.2021.00075

Danilo, N., Stefanović, D., Dakic, D., Sladojevic, S., & Ristic, S. (2021). *Analysis of the Tools for Static Code Analysis*. 1–6. https://doi.org/10.1109/INFOTEH51037.2021.9400688

Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., & Avgeriou, P. (2022). Can Clean New Code Reduce Technical Debt Density?. *IEEE Transactions on Software Engineering*, *48*(5), 1705–1721. https://doi.org/10.1109/TSE.2020.3032557

Dijkstra, E. W., Dahl, O. J., & Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

Fan, L., Su, T., Chen, S., Meng, G., Liu, Y., Xu, L., & Pu, G. (2018). Efficiently manifesting asynchronous programming errors in Android apps. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 486–497. https://doi.org/10.1145/3238147.3238170

Farhanaaz, & Sanju, V. (2016). An exploration on lexical analysis. *2016 International Conference on Electrical, Electronics, And Optimization Techniques (ICEEOT)*, 253–258. https://api.semanticscholar.org/CorpusID:11121418

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gupta, A. (2018). *Intelligent code reviews using deep learning*. https://api.semanticscholar.org/CorpusID:52219239

Haouari, D., Sahraoui, H., & Langlais, P. (2011). How Good is Your Comment? A Study of Comments in Java Programs. *2011 International Symposium on Empirical Software Engineering and Measurement*, 137–146. https://doi.org/10.1109/ESEM.2011.22

Herka, I. (2022, February). *Naming conventions in programming – a review of scientific literature — Makimo – Consultancy & Software Development Services*. makimo.com. https://makimo.com/blog/scientific-perspective-on-naming-in-programming/

Hippisley, A. (1976). Lexical Analysis. *Handbook of Natural Language Processing*. https://api.semanticscholar.org/CorpusID:64025642

Huang, Y., Jia, N., Zhou, Q., Chen, X., Xiong, Y., & Luo, X. (2018). Guiding developers to make informative commenting decisions in source code. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 260–261. https://doi.org/10.1145/3183440.3194960

Ilyas, B., & Elkhalifa, I. (2016). *Static Code Analysis: A Systematic Literature Review and an Industrial Survey*. https://api.semanticscholar.org/CorpusID:65258085

Jiang, T., Gradus, J. L., & Rosellini, A. J. (2020). Supervised Machine Learning: A Brief Primer. *Behavior Therapy*, *51*(5), 675–687. https://doi.org/10.1016/j.beth.2020.05.002

JingKai, S., Cuiyun, G., Lingling, F., Sen, C., & Yang, L. (2019). CORE: Automating Review Recommendation for Code Changes. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 284–295. https://api.semanticscholar.org/CorpusID:209439473

Keuning, H., Heeren, B., & Jeuring, J. (2021). A Tutoring System to Learn Code Refactoring. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 562–568. https://doi.org/10.1145/3408877.3432526

Keuning, H., Jeuring, J., & Heeren, B. (2023). A Systematic Mapping Study of Code Quality in Education. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 5–11. https://doi.org/10.1145/3587102.3588777

Kim, H., Kwon, Y., Joh, S., Kwon, H., Ryou, Y., & Kim, T. (2022). Understanding automated code review process and developer experience in industry. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1398–1407. https://doi.org/10.1145/3540250.3558950

Lee, G. (1982). *Syntax Analysis and Parsing Algorithms*. https://api.semanticscholar.org/CorpusID:63301130

Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., & Fontana, F. A. (2021). A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and

tools. *Journal of Systems and Software*, *171*, 110827–110828. https://doi.org/https://doi.org/10.1016/j.jss.2020.110827

Møller, A., & Schwartzbach, M. I. (2020). *Static Program Analysis*. Department of Computer Science, Aarhus University. https://cs.au.dk/~amoeller/spa/

Nachtigall, M., Nguyen Quang Do, L., & Bodden, E. (2019). Explaining Static Analysis - A Perspective. *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 29–32. https://doi.org/10.1109/ASEW.2019.00023

Ovchinnikova, E. (2012). Natural Language Understanding and World Knowledge. In *Integration of World Knowledge for Natural Language Understanding* (pp. 15–37). Atlantis Press. https://doi.org/10.2991/978-94-91216-53-4_2

Palomba, F., Bavota, G., Penta, M. D., Fasano, F., Oliveto, R., & Lucia, A. D. (2017). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, *23*, 1188–1221. https://api.semanticscholar.org/CorpusID:215767620

Pandey, A. K., Tripathi, A., Alenezi, M., Agrawal, A., Kumar, R., & Khan, R. A. (2020). A Framework for Producing Effective and Efficient Secure Code through Malware Analysis. *International Journal of Advanced Computer Science and Applications*, *11*(2). https://doi.org/10.14569/IJACSA.2020.0110263

Perera, R., & Nand, P. (2017). Recent Advances in Natural Language Generation: A Survey and Classification of the Empirical Literature. *Comput. Informatics*, *36*, 1–32. https://api.semanticscholar.org/CorpusID:263896381

Piater, J. (2005, January). *Formatting*. iis.uibk.ac.at. https://iis.uibk.ac.at/public/piater/courses/Coding-Style/ar01s03.html

Plosch, R., Gruber, H., Korner, C., & Saft, M. (2010). A Method for Continuous Code Quality Management Using Static Analysis. *2010 Seventh International Conference on the Quality of Information and Communications Technology*, 370–375. https://doi.org/10.1109/QUATIC.2010.68

Pratap, P. (2023). The evolution of computer programming languages. *International Journal of Advanced Research in Science, Communication and Technology*, *3*, 69–76. https://doi.org/10.48175/ijarsct-13110

Pressman, R. S., & Maxim, B. R. (2014). *Software engineering : a practitioner's approach*. Mcgraw-Hill Education.

Priyanka, M. (2024). Reinforcement Learning: A Comprehensive Overview. *International Journal of Innovative Research in Computer Science and Technology*. https://api.semanticscholar.org/CorpusID:269407468

Pugh, D. (2018, March). *A brief list of programming naming conventions*. deanpugh.com. https://www.deanpugh.com/a-brief-list-of-programming-naming-conventions

Qiao, Y., Wang, J., Cheng, C., Tang, W., Liang, P., Zhao, Y., & Li, B. (2024). Code Reviewer Recommendation Based on a Hypergraph with Multiplex Relationships. *Arxiv*. https://api.semanticscholar.org/CorpusID:267061002

Rani, P., Birrer, M., Panichella, S., Ghafari, M., & Nierstrasz, O. (2021). What Do Developers Discuss about Code Comments?. *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 153–164. https://doi.org/10.1109/SCAM52516.2021.00027

Rios, N., Mendonça, M., Seaman, C., & Spínola, R. (2019, August). *Causes and Effects of the Presence of Technical Debt in Agile Software Projects*.

Rodrigues, E., & Montecchi, L. (2019). *Towards a structured specification of coding conventions*. https://doi.org/10.1109/prdc47002.2019.00047

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach* (Third edition, Global edition). Pearson.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). Modern code review: a case study at google. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 181–190. https://doi.org/10.1145/3183519.3183525

Sas, D., & Avgeriou, P. (2020). Quality attribute trade-offs in the embedded systems industry: an exploratory case study. *Software Quality Journal*, *28*(2), 505–534. https://doi.org/10.1007/s11219-019-09478-x

Shaveta. (2023). A review on machine learning. *International Journal of Science and Research Archive*, *9*(1), 281–285. https://doi.org/10.30574/ijsra.2023.9.1.0410

Smit, M., Gergel, B., Hoover, H. J., & Stroulia, E. (2011, September). *Code convention adherence in evolving software*. IEEE Xplore. https://doi.org/10.1109/ICSM.2011.6080819

Sojer, M., Alexy, O., Kleinknecht, S., & Henkel, J. (2014). Understanding the Drivers of Unethical Programming Behavior: The Inappropriate Reuse of Internet-Accessible

Code. *Journal of Management Information Systems*. https://doi.org/10.1080/07421222.2014.995563

Stegeman, M., Barendsen, E., & Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 160–164. https://doi.org/10.1145/2999541.2999555

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and Why Your Code Starts to Smell Bad. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *1*, 403–414. https://doi.org/10.1109/ICSE.2015.59

Verma, R., Kumar, K., & Verma, H. K. (2023). Code smell prioritization in object-oriented software systems: A systematic literature review. *Journal of Software: Evolution and Process*, *35*(12), e2536. https://doi.org/https://doi.org/10.1002/smr.2536

Xue, Q. (2024). Unlocking the potential: A comprehensive exploration of large language models in natural language processing. *Applied and Computational Engineering*, *57*(1), 247–252. https://doi.org/10.54254/2755-2721/57/20241341

Zaytsev, V., & Bagge, A. H. (2014). Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, & E. Insfran (Eds.), *Model-Driven Engineering Languages and Systems* (Vol. 8767, pp. 50–67). Springer International Publishing. https://doi.org/10.1007/978-3-319-11653-2_4