RSET
RAJAGIRI SCHOOL OF
ENGINEERING & TECHNOLOGY
(AUTONOMOUS)

Project report on

CodeCleaner

Submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology

in

Computer Science and Engineering

By

Justin K A (U2103121)
Kannan M D (U2103123)
Kris Arun (U2103126)
Megha Krishna (U2103135)

Under the guidance of

Dr. Jincy J Fernandez

Department of Computer Science and Engineering
Rajagiri School of Engineering & Technology (Autonomous)
(Parent University: APJ Abdul Kalam Technological University)
Rajagiri Valley, Kakkanad, Kochi, 682039
April 2025

CERTIFICATE

This is to certify that the project report entitled ***"Code Cleaner"*** *is a bonafide record of the work done by* ***Justin K A (U2103121), Kannan M D (U2103123), Kris Arun (U2103126), Megha Krishna (U2103135)****, submitted to the Rajagiri School of Engineering & Technology (RSET) (Autonomous) in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology (B. Tech.) in "Computer Science and Engineering " during the academic year 2024-2025.*

Dr.Jincy J Fernandez
Project Guide
Associate Professor
Dept of CSE
RSET

Ms. Anu Maria Joykutty
Project Coordinator
Asst. Professor
Dept of CSE
RSET

Dr. Preetha K G
Professor & HOD
Dept of CSE
RSET

ACKNOWLEDGMENT

We wish to express our sincere gratitude towards **Dr Jaison Paul Mulerikkal CMI**, Principal of RSET, and **Dr. Preetha K G**, Head of the Department of Computer Science and Engineering for providing us with the opportunity to undertake our project, "Code Cleaner".

We are highly indebted to our project coordinators, **Ms. Anu Maria Joykutty**, Asst. Professor, Department of Computer Science and Engineering, **Dr. Jisha G**, Associate Professor, Department of Computer Science and Engineering, and **Dr Sminu Izudheen**, Associate Professor, Department of Computer Science and Engineering , for their valuable support.

It is indeed our pleasure and a moment of satisfaction for us to express our sincere gratitude to our project guide **Dr. Jincy J Fernandez** for her patience and all the priceless advice and wisdom she has shared with us.

Last but not the least, We would like to express our sincere gratitude towards all other teachers and friends for their continuous support and constructive ideas.

Justin K A

Kannan M D

Kris Arun

Megha Krishna

Abstract

In the realm of software development, maintaining readable, well-documented, and clean code is crucial for effective collaboration and long-term maintainability. However, achieving such standards can be time-consuming and challenging, especially for teams working under tight deadlines. The "**Code Cleaner**" project addresses this need by automating the refinement of code through advanced machine learning techniques and sophisticated algorithms.

This tool accepts code input, regardless of its initial quality, and processes it to produce highly readable, optimized, and well-commented code, adhering to clean coding practices. Leveraging natural language processing and code analysis, Code Cleaner reformats code to conform to standard style guides, ensuring consistency and readability. It intelligently inserts explanatory comments to enhance code understanding, optimizes performance by identifying and eliminating inefficiencies, and detects potential errors to improve code reliability. Code Cleaner is designed as a versatile tool for developers across various domains. By automating the enhancement of code, it saves time and effort, assisting developers in maintaining high-quality, professional codebases with ease.

This project represents a significant advancement in code quality enhancement, combining machine learning with practical coding insights to deliver clean, efficient, and well-documented code. By addressing the common challenges of code readability and maintainability, Code Cleaner aims to become an indispensable asset for programmers, fostering collaboration and adherence to best practices in software development.

Contents

List of Abbreviations

NLP - Natural Language Processing

AI - Artificial Intelligence

DSL - Domain-Specific Language

GPT-3 - Generative Pre-trained Transformer 3

ASAP - Automatic Semantic Augmentation of Prompts

BLEU - Bilingual Evaluation Understudy

T5 - Text-to-Text Transfer Transformer

CugLM - Code Understanding and Generation Language Model

READSUM - Retrieval-Augmented Adaptive Transformer for Source Code Summarization

AST - Abstract Syntax Tree

DRD - Daily Refactoring Density

WRD - Weekly Refactoring Density

DTW - Dynamic Time Warping

CSD - Code Smell Difference

ECS - End-stage Code Smells

ELC - End-stage Lines of Code

ICS - Initial-stage Code Smells

ILC - Initial-stage Lines of Code

CC - Code Churn

UI - User Interface

API - Application Programming Interface

DOM - Document Object Model

IDE - Integrated Development Environment

CI/CD - Continuous Integration/Continuous Deployment

List of Figures

List of Tables

Chapter 1

Introduction

This project aims to drive significant improvements in key areas, including generating comprehensive documentation, renaming variables for greater clarity, and promoting standardized coding practices. CodeCleaner is particularly useful for batch processing, meaning that you could use it when you want to polish up giant code-bases. An advanced natural language processor with a myriad machine learning models would be a great supplement to make the CodeCleaner an optimizing tool for programmers dealing with complicated codebases as well as for document generation.

1.1 Background

Code is required to be read, maintained, and comply with best practices: all things that are required in any effective development in an increasingly rapidly changing software environment.

Maintaining a high-quality code becomes difficult when projects are large and complex. Further entrenching the challenges for modern developers, legacy codebases are usually produced without modern specifications. The main solutions to these problems are automation of vital code enhancement activities, such as documentation generation, renaming variables, and improving readability. To this end, CodeCleaner uses NLP and machine learning to make these easy tasks for developers while making the migration of legacy codes to new standards less painful.

1.2 Problem Definition

The complexity and inefficiency of manually reviewing, refactoring, and documenting code, especially in large and outdated codebases is very high. Manually ensuring consistency and readability across code can be time-consuming, error-prone, and difficult to

scale, impacting maintainability and software quality.

1.3 Scope and Motivation

Scope:

CodeCleaner scope allows it to automatically improve code quality by batch processes that include generating documentation, renaming variables, and refactoring for better readability. In a single solution across different programming languages, it is designed to scale to allow standards and maintenance in bigger projects or legacy codebases. CodeCleaner uses intelligent machine learning techniques to step in for consistency and enforce coding standards. Such characteristics make it best apt for use in enterprise systems where write-clarity, maintainability, and bulk processing capability, are expected of it, stand by it towards hassle-free handling of massive codebases.

Motivation:

The motivation to grow CodeCleaner is certainly from the fact that there has been an increasing need for such tools, which will help reduce the need to maintain code, especially since this need has increased due to scaling and dynamic evolution of codebases. Manual code refactoring and documentation often involve mistakes and considerably slow down the development. CodeCleaner addresses these issues where maintenance takes decently more time and effort for upholding good code standards. It also helps developers get an introductory understanding of inherited or legacy codebases faster by improving readability and documentation. Thus, it is desired that CodeCleaner creates an environment where well-organized, clear, and consistent code significantly stimulates efficiently accelerating long-term scalability.

1.4 Objectives

- Constructing a means to enhance the clarity and manageability of previously written code through automated refactoring.
- Intelligent generation of documentation on codes for vie understanding in function or structures of codes.
- Implement automation for identifying and renaming ambiguous variable names

through machine learning.

- Develop a means of improving the clarity and manageability of previous code bases through automated refactoring.
- Integrated with intelligent machine learning algorithms to facilitate automation in identifying and renaming ambiguous variable names.
- Get a batch processing facility for powerful flexibly handling immense amounts of code without real-time limits. Standardized codes must further encourage well-coding practices with the help of developers.

1.5 Challenges

There are many challenges that CodeCleaner faces, balancing between automating and keeping all the possible accuracy while not losing the intent of the original code. Further, it is quite complicated to ensure that machine learning models generate relevant documentation without misinterpretation as well as adding inaccuracies.
CodeCleaner must also be made adaptable to changing coding languages and standards, which would pose technical challenges, in addition to inventing an efficient batch-processing system capable of handling really large codebases with no performance issues.

1.6 Assumptions

1. The premise of this project is that the input code is standard syntax-wise and structurally according to its respective programming language to process and generate from it documentation.
2. By what the model assumes about the training datasets, they represent typical codebases so that one can generalize across many programming styles.
3. CodeCleaner expects that project end-users have some basic knowledge about coding because they're supposed to judge or at least know how to make the changes suggested.
4. The tool assumes enough computer resources would be available to do model training and batch processing without major interruptions.

1.7 Societal / Industrial Relevance

Industrial Relevance: CodeCleaner is really a very important tool in industry for improving code quality, readability, and maintainability and thus increasing efficiency in software development. This can be more useful to large companies that deal with legacy codes, where consistent well-documented codes can lead to a reduced time and resources used during maintenance processes and updates.

CodeCleaner does exactly this: it takes the document and the code and works on it so that it becomes more automated and refined, freeing the developer to do innovative doing and ultimately increasing productivity and the life of the code itself.

Societal Relevance: In the greater programming community, CodeCleaner stands for the case for better code, especially in that it means cleaner more understandable code that is better shareable and reusable.

Such situations are most valued by open-source projects, as here code readability means the difference between collaborative effort and lone ranger activity. Moreover, better quality still results in better software creations that might end up affecting positively people and communities around the world.

1.8 Organization of the Report

This report provides a comprehensive overview of the CodeCleaner project, starting with a detailed introduction to its objectives, background, and problem definition. The following chapters will focus on the methodologies applied, including an analysis of machine learning and NLP techniques integrated into CodeCleaner. We wil also covers the literature review, referencing relevant research papers and comparing CodeCleaner's approach to existing code-quality enhancement methods. Subsequent chapters discuss the technical implementation, testing, and evaluation. The report concludes by highlighting the project's societal and industrial relevance, along with future development possibilities.

1.9 Conclusion

In summary, CodeCleaner tries to capture the colossal challenge of improving readable, maintainable and consanguinous code across disparate codebases. It aims to give au-

tomated solutions in code documentation, variable renaming, and overall code quality improvement. In that respect, the project turns out to be equally important for industry towards efficient software maintenance as well as for society by promoting better programming practices. Clear and strong objectives have factored in making it precocious development for future developers in course of managing abstraction or old code.

Chapter 2

Literature Survey

2.1 Refactoring Techniques to Improve Software Quality

Refactoring is a way of restructuring the code internally without changing the behavior towards outside, and has value in making software easily maintainable. Such refactoring takes place for the "bad smells" which denote design choices made poorly, like duplicated code, large classes, which increase complexity for all involved. Its purpose is to eliminate such issues, and even lead it toward a reduction of technical debt while simplifying structures within the code for better readability and easier maintenance [1].

2.1.1 Identifying and Addressing Bad Smells

Code smell detection tools like JDeodorant identify areas requiring improvement, including issues such as Large Class, Long Method, Duplicate Code, and Feature Envy. To address these, refactoring techniques are systematically applied:

- **Large Class**: Breaks down complex classes to enhance comprehension.
- **Long Method**: Shortens lengthy methods to improve readability and debugging.
- **Duplicate Code**: Removes redundant code to reduce maintenance complexity.
- **Feature Envy**: Relocates methods to appropriate classes to improve cohesion.

Modern IDEs provide automated tools that streamline the identification and elimination of these issues. By integrating AI-driven predictive models, refactoring practices can further evolve into proactive and adaptive systems that ensure long-term code maintainability.

2.2 Semantic Programming by Example with Pre-Trained Models

This paper introduces a novel integration of pre-trained language models and inductive synthesis frameworks to enhance the automation of semantic and syntactic transformations in programming by example. Traditional inductive synthesizers are limited to syntactic manipulations and require extensive Domain-Specific Languages (DSLs) to encode semantic transformations. By leveraging pre-trained large language models like GPT-3, the proposed framework introduces semantic operators that complement inductive synthesis, enabling a broader range of tasks, such as mixed syntactic-semantic transformations, string profiling, and code refactoring[2].

2.2.0.1 Overview of Methodology

The integration operates through the following key components:

- **Semantic Operators**: Learnable operators capable of performing tasks such as semantic lookup, position extraction, and condition evaluation.
- **Deferred Query Execution**: A mechanism to minimize expensive queries to language models during the learning process.
- **Inductive Synthesis Framework**: Extends DSLs with semantic operators while leveraging deductive backpropagation to decompose problems into subproblems solvable by either syntactic or semantic methods.

2.2.1 Semantic Operators

The framework introduces three types of semantic operators, each tailored to a specific task:

1. **Semantic Map (SemMap)**: Allows semantic lookups for input strings based on a few-shot learning approach using pre-trained models.
2. **Semantic Positioning (SemPos)**: Identifies relevant positions in input strings, enabling targeted transformations such as substring extraction.
3. **Semantic Conditions (SemCond)**: Performs classification tasks, e.g., distinguishing data types or formats.

Each operator interacts with the language model by constructing a prompt and parsing the response to execute transformations.

2.2.2 Deferred Query Execution

This framework comprises the delayed execution paradigm for alleviating some of the high costs attached to querying large language models. During the period of training, semantic operators are made to act like oracles while are usually guided in the synthesizing process through a ranking mechanism, which selects that combination of operators that is most efficient. This reduces the number of queries by a large margin for the purpose of program synthesis.

2.2.3 Integration with DSLs

The framework has added semantic enrichment to the existing DSLs such as FlashFill. These operators will be directly embedded in the process of inductive synthesis, enabling the model to make both syntactic and semantic transformations. The modified DSL is now capable of generating sentences for grammar exercise, refactoring codes, and string profiling among other things.

2.2.3.1 Ranking and Learning

Programs are ranked based on:

- Preference for syntactic over semantic operators.
- Minimal usage of distinct queries to the language model.
- Compactness of the generated program.

The ranking ensures efficient program execution and prioritizes solutions that rely on deterministic rules.

2.2.4 Applications and Evaluation

The framework demonstrates it's utility across several domains:

- **String Transformations**: Automates tasks such as date formatting and grammatical exercise generation.

- **Code Refactoring**: Enhances tools like Blue-Pencil by introducing semantic understanding for variable renaming and documentation.
- **Profiling**: Learns succinct patterns to describe string datasets, improving data quality and program robustness.

Experiments reveal that combining syntactic and semantic operators reduces the number of required examples and enhances task generalization.

2.2.5 Key Contributions

The paper highlights the following contributions:

- A fresh new framework which joins pre-trained language models with inductive synthesis.
- Deferred query execution to learn efficiently.
- semantic operators deployed for wider application domains.
- Empirical assessment showing the changed performance in string transformation and profiling tasks.

2.3 Enhancing Code Summarization with Semantic Augmentation

The Automatic Semantic Augmentation of Prompts (ASAP) method involves automatic semantic enrichment of prompts to better summarize code through their inclusion of semantic elements such as function names, data flow, and tagged identifiers. This allows the language model to amass richer context that will enhance the understanding of the code with regard to performace on summarization tasks[3].

2.3.1 Semantic Prompt Augmentation in Code Summarization

ASAP improves few-shot learning by integrating relevant semantic facts into prompts, allowing the model to generate more accurate and coherent summaries. Key components of the ASAP approach include:

- **BM25 Retrieval Algorithm**: Selects relevant code-comment examples to optimize few-shot learning.

- **Semantic Facts Augmentation**: Adds information like function signatures, data flow graphs, and identifier tags to enhance prompt quality.
- **CodeSearchNet Dataset**: Provides a diverse dataset of code-comment pairs across multiple programming languages for evaluation.

Empirical results demonstrate that ASAP achieves improved BLEU (Bilingual Evaluation Understudy) scores across languages, which is a metric used to evaluate the quality of machine-generated text. Thus it paves the way for scalable, automated documentation tools.

2.4 Automated Variable Renaming: Are We There Yet?

This paper investigates the efficiency of data-driven techniques for employing automated variable renaming. The work analyzes three different models: an n-gram cached language model[4], the T5 model[5], and the CugLM model[6]. These are trained and evaluated on different datasets, and the coverage assesses their ability to provide developers with meaningful identifiers that match their choices[7].

2.4.1 Overview of Methodology

The study involves three core approaches:

- **N-Gram Cached Model**
- **Text-to-Text Transfer Transformer (T5)**
- **CugLM: Transformer-Based Model**

Each method is trained and evaluated using three datasets designed to assess their performance and robustness.

2.4.1.1 N-Gram Cached Model

An N-Gram Cached Model is a probabilistic language model that uses N-grams (sequences of N words) to predict the next word in a sequence. This approach relies on statistical analysis of word co-occurrence to estimate the probability of a word appearing given its context.The N-Gram Model uses the previous tokens sequence to predict the next token, and caching hierarchical context improves scoring.

Advantages:

- Provides a lightweight, efficient solution.
- Performs well when the code is locally repetitive.

Limitations:

- Cannot handle rare tokens due to the vocabulary constraint.
- Unable to capture long-range dependencies.

By defining full token probability in terms of all preceding tokens, the model strengthens the probabilities with local context by being dynamic in adaptation.

2.4.1.2 Text-to-Text Transfer Transformer (T5)

T5 (Text-to-Text Transfer Transformer) is a transformer-based model developed by Google, designed to handle any NLP task in a unified text-to-text format. Instead of treating tasks separately, T5 converts everything into a text generation problem. T5 is adapted to rename variables by training it to predict masked tokens in code.

Pre-Training:

- Randomly masks tokens in code during training to enhance the model's understanding of syntax and semantics.

Fine-Tuning:

- Methods are masked, and the model predicts meaningful replacements based on surrounding context.

Performance Enhancements: The T5 model applies a unified text-to-text framework and uses adaptive learning rates during fine-tuning to optimize for specific tasks.

2.4.1.3 CugLM: Transformer-Based Model for Code Completion

CugLM (short for Code Understanding and Generation Language Model) is a transformer-based model designed specifically for code analysis and generation tasks. It is optimized for code refactoring, bug detection, and automatic code generation. CugLM specializes in identifier prediction, employing three pre-training tasks:

1. **Masked Token Prediction:** Focuses on predicting missing identifiers in code.
2. **Code Continuation:** Determines if two code fragments follow logically.
3. **Left-to-Right Language Modeling:** Predicts the next token in a sequence.

Multi-Task Learning: The fine-tuning process integrates identifier type prediction, enhancing its effectiveness in renaming variables.

Advantages:

- Outperforms other models in large-scale tests.
- High accuracy for common identifiers.

Challenges:

- Limited by a fixed vocabulary size of 50k tokens.
- Fails to generalize well for rare identifiers.

2.4.2 Dataset Design and Use

The study builds three datasets:

- **Large-Scale Dataset:** Comprises millions of instances of Java methods for training.
- **Reviewed Dataset:** Focuses on identifiers introduced or modified during code reviews.
- **Developers' Dataset:** Extracted from real-world refactoring commits.

Each dataset is curated to represent varying levels of identifier quality and testing complexity.

2.4.3 Performance Metrics and Evaluation

Correct Prediction: A prediction is correct if it matches the identifier chosen by the developer.

Confidence Levels: Higher confidence predictions are strongly correlated with correctness, particularly for the CugLM model. The study also introduces "complete-match" and "partial-match" evaluation heuristics for models that predict multiple tokens.

2.4.4 Key Findings

- **CugLM** consistently outperforms T5 and n-gram models, particularly for identifiers with high training set occurrence.
- Confidence in predictions can act as a proxy for quality, aiding in the development of recommendation tools.
- Complex identifiers composed of multiple words pose challenges across all models.

Practical Implications: The findings suggest that data-driven models, particularly transformer-based approaches, are ready to be integrated into rename refactoring tools, provided they incorporate confidence thresholds and contextual information.

Future Directions:

- Expanding vocabulary constraints for CugLM using advanced tokenization strategies.
- Integrating naming conventions and contextual cues into prediction mechanisms.

2.5 READSUM: Transformer-Based Source Code Summarization

The paper proposed by Choi, Na, Kim and Lee introduces a novel model, READSUM, designed to automatically generate high-quality code summaries by integrating retrieval-augmented inputs and adaptive attention mechanisms. The model effectively captures sequential and structural aspects of source code, leveraging embedding augmentation,

adaptive transformer encoders, and a dual copy mechanism. These components are described below[8].

2.5.0.1 Overview of Methodology

The READSUM model referred in Fig 2.1 operates in four distinct phases:

- Embedding Augmentation Layer
- Transformer Encoders for Code and Summaries
- Summary Transformer Decoder
- Dual Copying Mechanism

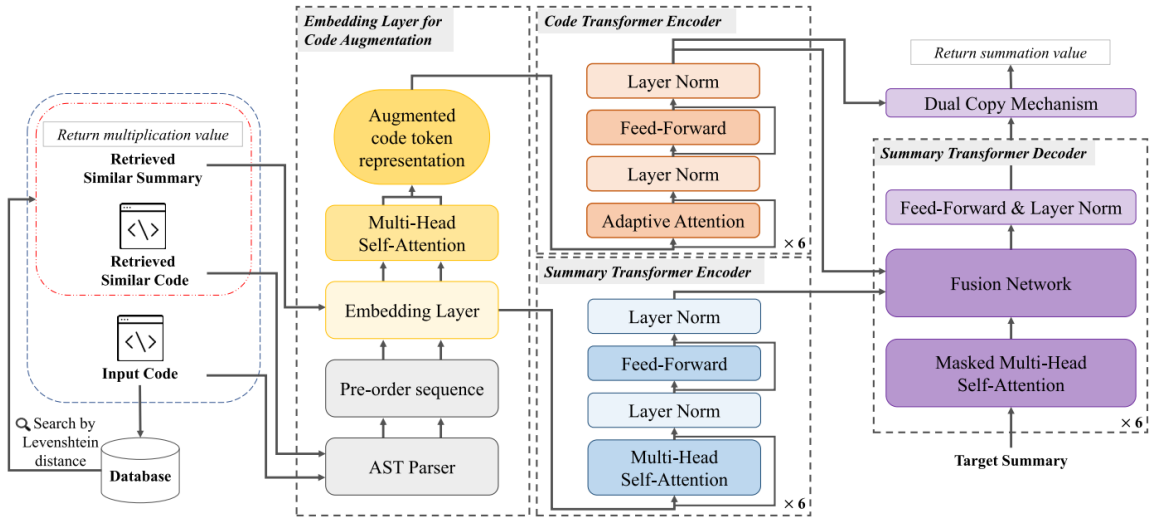


Figure 2.1: Diagram of the READSUM model architecture.

2.5.1 Embedding Augmentation Layer

The model begins by processing both the original code tokens and retrieved similar tokens to capture contextual relevance.

Multi-Head Self-Attention: Relationships between the original and retrieved tokens are identified using multi-head self-attention.

Augmented Representations: The original tokens are augmented with relevant retrieved tokens to create a comprehensive representation:

$$A_{\text{rel}} = \text{MultiAtt}(E_n, E_{n'}) \tag{2.1}$$

$$E_{\text{aug}} = E_n + z \cdot A_{\text{rel}} \tag{2.2}$$

Here, z indicates relevance score, E_n represents embeddings of the original code, and $E_{n'}$ represents embeddings of the retrieved tokens.

2.5.2 Code and Summary Transformer Encoders

The READSUM model uses two transformer encoders: one for the augmented code and the other for retrieved summaries. These allow the model to independently process code structure and summary context.

Code Transformer Encoder Adaptive Attention Mechanism: The traditional self-attention mechanism is enhanced with a trainable bias term, accounting for distance-based relevance in AST nodes:

$$\text{AdapAtt}(Q, K, V, d) = \text{softmax}(QK^T + g(d))V \tag{2.3}$$

Here, Q (Query), K (Key), and V (Value) are standard components of the self-attention mechanism in transformers. Q represents the current token, V holds the actual information from the tokens, K represents all tokens that can be "searched" to determine their relevance to the query, d represents AST node distances, and $g(d)$ applies a Gaussian-based bias for structural weighting.

Role of Adaptive Attention: Lower layers focus on local code structure, emphasizing nearby nodes, while upper layers capture broader dependencies across the code.

Summary Transformer Encoder Retrieved summaries are encoded using standard transformer layers, capturing context relevant to similar code summaries.

2.5.3 Summary Transformer Decoder

The decoder integrates outputs from both encoders to generate the final summary.

Fusion Network: Attention scores from the encoders are blended using a similarity-based weighting:

$$\alpha_1 = \text{softmax}(QK_1^T), \quad \alpha_2 = \text{softmax}(QK_2^T) \tag{2.4}$$

$$\text{FusAtt} = \frac{1}{1+z} \cdot \alpha_1 V_1 + z \cdot \alpha_2 V_2 \tag{2.5}$$

The Fusion Network manifests a critical dependence on the relative combination of contextual information from both the code encoder as well as the summary encoder, thus producing effectively accurate and coherent summaries. Through calculation of attention scores as in Eq 2.4 (α1,α2) from both sources and blending through the weighting mechanism (z), the network can ensure that the decoding side accesses a balanced mix of relevant information. So under this scheme, the model could vert both attention between the relevant code-based context and the summary context, depending on the requirements of the task. The outcome would result in the fused attention vector (FusAtt), which would provide the necessary consolidated representation such that it can ultimately increase the quality of the decoded outputs into rich contextual and semantically meaningful representations based on their application to generated summary quality as in Eq 2.5.

2.5.4 Dual Copying Mechanism

To enhance information retention, the model uses a dual copy mechanism that directly incorporates tokens from the input code and retrieved summaries.

Prediction of Final Summary: At timestep t, the summary is predicted using a combination of attention scores and prediction probabilities:

$$P_{\text{dual}}(w) = \lambda_1 \alpha_1 + \lambda_2 \alpha_2 + \lambda_3 p_t \tag{2.6}$$

where λ_1, λ_2, and λ_3 are learnable weights optimized to balance contributions from code, summaries, and predictions, as given in Eq 2.6.

Through these stages, READSUM achieves a seamless integration of abstractive and extractive techniques, setting a high benchmark for code summarization. The use of adaptive attention and the dual copy mechanism underscores the model's innovative capabilities.

2.6 Empirical Study of Refactoring Rhythms and Tactics in Software Development

This paper presents an empirical investigation into the patterns of refactoring rhythms and tactics used in software development, with a focus on their impact on code quality. By analyzing data from 196 Apache projects, the study introduces metrics and techniques to uncover rhythms, tactics, and their relationships with code quality [9].

2.6.1 Overview of Methodology

The methodology employed in this study can be broadly categorized into the following steps:

- **Data Collection and Preprocessing:** The dataset used was extracted from the 20-MAD Apache dataset, which includes commit histories and issue data for over 700 projects. Java projects were filtered based on criteria such as commit count, lifespan, and code composition.
- **Defining Refactoring Metrics:** Two new metrics, Daily Refactoring Density (DRD)[9] and Weekly Refactoring Density (WRD)[9], were introduced to quantify refactoring activities at daily and weekly levels.
- **Refactoring Rhythm Identification:** The study categorized refactoring rhythms into work-day and all-day based on refactoring density distributions across days of the week. Statistical methods like the Kruskal-Wallis test were employed to determine rhythm patterns.
- **Clustering Refactoring Tactics:** Using Dynamic Time Warping (DTW), the researchers identified four refactoring tactics: Intermittent Spiked Floss, Frequent Spiked Floss, Intermittent Root Canal, and Frequent Root Canal.
- **Impact on Code Quality:** Code smells were used as a proxy for code quality, with changes tracked across project stages. The Scott-Knott-ESD test[10] was employed to analyze the relationship between refactoring patterns and code quality metrics.

2.6.1.1 Findings

- **Refactoring Rhythms:**
  - Projects typically follow one of two rhythms:
    1. *Work-Day Refactoring:* Higher activity on weekdays.
    2. *All-Day Refactoring:* Uniform activity throughout the week.
  - Rhythms were found to be influenced by project and author profiles but did not significantly correlate with overall code quality.
- **Refactoring Tactics:**
  - Tactics were classified as:
    1. *Floss-Based:* Frequent and consistent refactoring alongside regular development.
    2. *Root Canal-Based:* Focused, infrequent refactoring targeting design improvements.
  - Root canal-based tactics were associated with better code quality, as they were linked to reductions in code smells.
- **Code Quality Impacts:**
  - Floss-based tactics often correlated with increased code smells due to their integration with daily tasks.
  - Root canal-based tactics, on the other hand, were more effective in reducing design flaws and improving maintainability.

2.6.1.2 Metrics and Equations

- **Daily Refactoring Density (DRD):** This metric measures the proportion of refactoring activities in relation to the total code changes (churn) made on a specific day.

$$\text{DRD}(i) = \frac{\text{Refactoring Churn of the Day(i)}}{\text{Total Code Churn of the Day(i)}} \tag{2.7}$$

- **Weekly Refactoring Density (WRD):** This metric extends the concept of DRD to a weekly scale, providing insight into the frequency of refactoring activities over a longer time frame.

$$\text{WRD}(i) = \frac{\text{Refactoring Churn of the Week(i)}}{\text{Code Churn of the Week(i)}} \tag{2.8}$$

- **Code Smell Difference (CSD):** This metric evaluates the improvement in code quality by comparing the density of code smells before and after refactoring, normalized by the code churn.

$$\text{CSD}(i) = \frac{(\text{ECS(i)}/\text{ELC(i)}) - (\text{ICS(i)}/\text{ILC(i)})}{\text{CC(i)}/\text{ELC(i)}} \tag{2.9}$$

  where:

  - **ECS(i):** End-stage code smells, representing the number of code smells detected after refactoring at stage i.
  - **ELC(i):** End-stage lines of code, indicating the total lines of code present after refactoring at stage i.
  - **ICS(i):** Initial-stage code smells, representing the number of code smells detected before refactoring at stage i.
  - **ILC(i):** Initial-stage lines of code, indicating the total lines of code present before refactoring at stage i.
  - **CC(i):** Code churn, measuring the total number of lines of code added, modified, or deleted during the refactoring process at stage i.

2.6.2 Implications for Software Development

This study offers significant insights for developers and teams aiming to improve software quality. By understanding refactoring rhythms and tactics, practitioners can:

- Select appropriate refactoring strategies for different project stages.
- Balance daily development needs with targeted, high-quality refactoring efforts.
- Use rhythm and tactic insights to develop tooling that supports efficient refactoring practices.

Through its focus on rhythms, tactics, and code smells, this paper sets the stage for future research into automated tools and strategies to enhance software maintainability.

2.7 Summary and Gaps Identified

The literature review reveals a range of approaches to improving software code quality through refactoring, semantic programming, and code summarization. Refactoring primarily enhances readability and maintainability by addressing structural "bad smells" in code. Semantic programming integrates pre-trained models with inductive synthesis to automate tasks that require both syntactic and semantic comprehension, proving effective in repetitive tasks such as code restructuring. Additionally, augmenting prompts with semantic context has shown to improve code summarization accuracy by large language models. However as seen in Table 2.1, each approach presents limitations, including dependency on computational resources, lack of contextual depth in refactoring, and scalability challenges. The following table summarizes the advantages and disadvantages of each approach, followed by identified gaps in the current state of the art.

2.7.1 Summary of Literature Review

Table 2.1: Summary of Literature Review

Title	Advantages	Disadvantages
Code Refactoring Impact on Software Quality[1]	• Enhances code readability and maintainability. • Reduces technical debt by addressing code "bad smells".	• Primarily limited to syntax-based refactoring; lacks contextual insights for complex code structures. • Manual refactoring is error-prone and time-intensive.
Semantic Programming by Example with Pre-trained Models[2]	• Leverages semantic understanding for complex tasks like variable renaming and restructuring. • Automates repetitive code transformation tasks effectively.	• Highly dependent on computationally intensive language models. • Limited in handling syntactic tasks without manual integration of operators.

Automatic Semantic Augmentation of Prompts for Code Summarization[3]	• Increases language model accuracy by embedding semantic facts. • Improves code summarization quality and alignment with human expectations.	• Performance heavily relies on prompt engineering and suitable datasets. • Limited by the availability of high-quality semantic annotations for all code contexts.
READSUM: Transformer-Based Source Code Summarization[8]	• Combines retrieval-augmented inputs and adaptive attention mechanisms to enhance summarization accuracy and context relevance. • Adaptive attention captures both structural and sequential information, improving summarization for complex codebases.	• Dependence on retrieved similar code can limit performance for unique or novel code snippets. • The fusion process and dual copying mechanism increase implementation complexity and resource usage.

Empirical Study of Refactoring Rhythms and Tactics in Software Development[9]	• Provides insights into effective refactoring strategies, such as root canal tactics, which improve maintainability by reducing code smells. • Metrics like Daily Refactoring Density (DRD) and Code Smell Difference (CSD) can inform better automated refactoring tools.	• Focuses primarily on behavioral patterns (rhythms and tactics) rather than automated refactoring implementations. • Limited application for automating variable renaming or summarization due to its emphasis on refactoring frequency metrics.

Chapter 3

System Requirements

This section outlines the essential specifications for the **CodeCleaner** project to ensure optimal performance in analyzing, refactoring, and improving code quality. By leveraging advanced machine learning models like **GPT-4o mini, CodeT5-base, and DeepSeek Coder 6.7B**, the project aims to streamline code optimization, enhance readability, and maintain best coding practices. This section provides a structured framework for developing a robust and efficient AI-powered code refinement tool.

3.1 Hardware and Software Requirements

While the **CodeCleaner** project does not require dedicated hardware, a capable computing environment is essential to run machine learning models effectively. Below are the recommended hardware and software requirements.

3.1.1 Hardware Requirements

- **I**ntel Core i5 processor or higher for efficient processing.
- **A** graphics card with machine learning capabilities, such as an NVIDIA RTX series GPU, to accelerate model training and inference.
- **A** minimum of 8 GB RAM to handle large datasets and model operations effectively.
- **A**t least 50 GB of free storage space to store datasets, trained models, and software dependencies.
- **S**tandard peripheral devices, including a screen, keyboard, and mouse, for user interaction.
- **S**table internet connection for API calls, cloud-based processing, and updates

3.1.2 Software Requirements

- **Development Environment:** Visual Studio Code is a versatile code editor for writing and managing the project code.
- **Front-end Framework:** Either Flask for a lightweight backend with embedded front-end capabilities or React for a modern, dynamic user interface.
- **Browser/Terminal:** A web browser or a system terminal for testing and deploying the application .
- **Machine Learning Models:** Pre-trained models like DeepSeek-Coder:6.7B and CodeT5 for code summarization, refactoring, and other functionalities.

3.2 Functional Requirements

3.2.1 Code Analysis and Refactoring

CodeCleaner is designed to analyze and improve code quality by identifying redundant or inefficient. The system applies transformations to enhance structure and readability. Additionally, it suggests optimal variable names and function modularization to maintain clarity and efficiency in software development.

3.2.2 Indentation

Ensuring error-free code is a crucial functionality of CodeCleaner. The system automatically detects indentation errors by tracking braces and provides corrective suggestions, helping developers prevent potential runtime failures. By adhering to industry coding standards, CodeCleaner ensures that the generated code remains consistent, efficient, and maintainable.

3.2.3 Code Optimization and Performance Enhancement

Performance is a key factor in software development, and CodeCleaner assists in optimizing code execution. The system reduces redundant computations, leading to better performance and improved space complexity.

3.2.4 Documentation and Code Commenting

Well-documented code is essential for maintainability and collaboration. CodeCleaner generates detailed inline comments for complex logic, making the code easier to understand for developers. Furthermore, it provides summary documentation for functions and classes, ensuring clarity in software structure. This automated documentation process facilitates better teamwork and long-term project sustainability.

3.3 Summary of the Chapter

This chapter detailed the necessary hardware, software, and functional requirements for the CodeCleaner project, ensuring smooth operation and high efficiency in code analysis and optimization. With cutting-edge AI models, a robust development environment, and a user-friendly interface, CodeCleaner is designed to revolutionize code refactoring by enhancing readability, efficiency, and maintainability. By prioritizing automation, feedback-driven refinement, and seamless user interaction, this system aims to support developers in writing cleaner, high-quality code with minimal effort.

Chapter 4

System Design

The System Design chapter outlines the foundational blueprint of the CodeCleaner project, focusing on the architectural, algorithmic, and technological aspects necessary for its successful implementation. It provides a comprehensive view of how the system will be structured, the components involved, and their interactions. This chapter aims to bridge the gap between conceptual planning and practical execution, offering detailed insights into the design and functionality of individual modules. By defining the system's architecture, component interactions, data flow, and tools, this chapter sets the groundwork for efficient development and integration. Additionally, it includes key deliverables, datasets, and a project timeline to ensure a systematic approach toward achieving the project objectives.

4.1 System Architecture

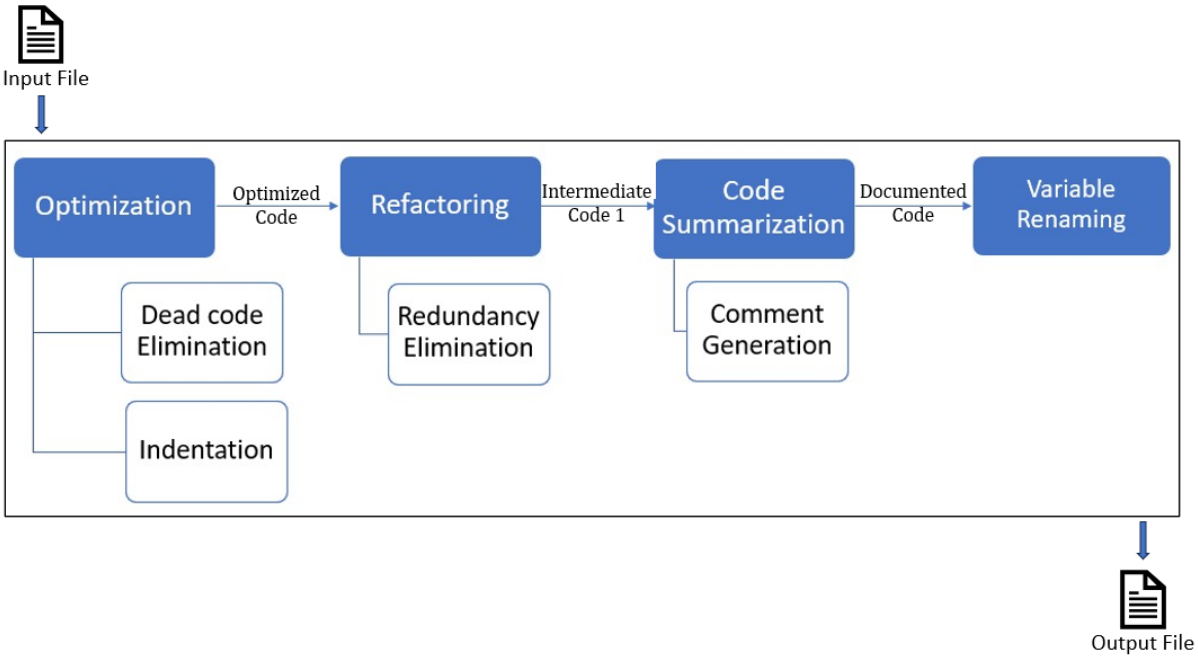


Figure 4.1: Architecture Diagram

The CodeCleaner pipeline, as depicted in Figure 4.1, outlines a structured process for improving the quality, readability, and maintainability of source code. This workflow consists of several sequential stages, each designed to refine different aspects of the code before producing an optimized output file. The process begins with an input file that contains the raw source code. The first stage is **Optimization**, which focuses on removing unnecessary elements from the code. Optimization includes:

Dead Code Elimination: Removing code segments that are never executed or no longer contribute to the program's logic.

Unused Variable Elimination: Identifying and removing declared variables that are never used, helping to reduce memory usage and improve readability.

Optimized code is passed to the next phase, which is **Refactoring**. It focuses on improving the structure and organization of the code without changing its functionality. This stage includes:

Redundancy Elimination: Removing duplicated or unnecessary computations to make the code more efficient.

Indentation: Ensuring proper formatting and spacing for better readability, following best coding practices.

First intermediate code is given for **Code Summarization**, where meaningful comments are generated to improve documentation. Final stage is **Variable Renaming**, which improve the clarity of the code by renaming variables to reflect their purpose more accurately.

4.2 Algorithm Design

Algorithm 1 Code Refactoring Algorithm

Require: Java code file

Ensure: Refactored code with function calls and definitions

1: Read the Java code file line by line.
2: Compare every line with subsequent lines to find matching blocks.
3: **if** a repeated block is found **then**
4: Assign it a unique function name.
5: Store the block and function name.
6: **end if**
7: Traverse the code and replace each occurrence of repeated blocks with a function call.
8: Create new functions for each repeated block and append them to the refactored code.
9: Save the refactored code with function calls and function definitions.

Algorithm 2 Unreachable Code and Code Indentation Algorithm

Require: Java code file

Ensure: Cleaned and formatted code

Read the Java code file line by line.

Remove Unreachable Code:

Initialize a flag `is_reachable` to `True`.

for each line in the file **do**

    if the line starts a block with { **then**

        Set `is_reachable` to `True`.

    else if the line contains `return;` **then**

        Set `is_reachable` to `False`.

        Skip subsequent lines until a closing brace } is encountered.

    end if

    Include the line in the output if `is_reachable` is `True` or the line is a closing brace }.

end for

Adjust Indent Levels:

Track indentation level, increasing it for lines ending with {.

Decrease indentation level for lines starting with }.

Apply consistent spacing (e.g., 4 spaces) based on the indentation level.

Save the cleaned and formatted code to the output file.

4.3 USE CASE Diagram

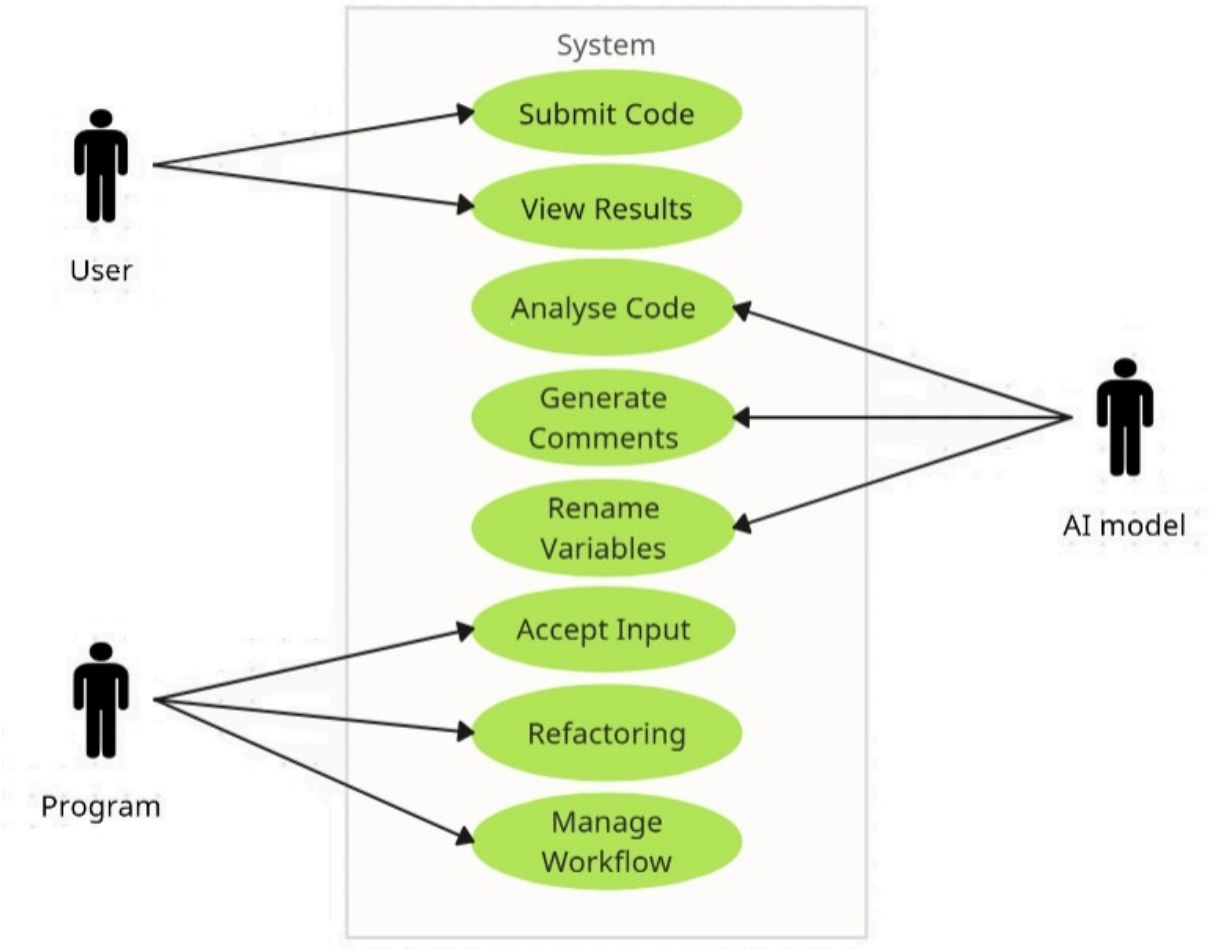


Figure 4.2: Use case diagram

4.4 Dataset Identified

The data set utilized for the CodeCleaner project is **CodeXGLUE**, a benchmark dataset for various code-related tasks, including code summarization(dataset found under the Code-to-Text subfolder). CodeXGLUE is a dataset created specifically to be a benchmark dataset for models trained in code-text related tasks.

It provides a vast set of code snippets consisting of multiple programming languages and their corresponding summaries, making it an ideal choice for training and fine-tuning models like DeepSeek-Coder and CodeT5. Using the CodeXGLUE dataset, CodeCleaner ensures access to high-quality and diverse examples, enabling the generation of accurate

and contextually relevant code summaries.

4.5 Methods- Module Division

This section describes the systematic approach undertaken to implement and execute the core modules of the **CodeCleaner** project: code optimization, code refactoring, code understanding and analysis (by way of code commenting) and variable renaming. While each module makes changes for bettering the readability, maintainability, and efficiency, all the while none-of these changes altering the functionality of the code . By breaking down the functionality of each module, this section highlights the step-by-step processes—from semantic analysis for generating comments to identifying and removing redundant code blocks—used to achieve an automated, effective code refinement tool.

4.5.1 Code Optimization

The primary purpose of this module is to optimize codes for performance by eliminating the dead codes and reducing the complexity of codes. It begins with reachability analysis, where the tool discovers and marks as unreachable any segment of code thereafter following a return as it pinpoints the existence of dead code that could be excluded. Next, each variable and method declared within reachable code is checked to find out which ones remain unutilized, and the ones used- the unused code is altogether eliminated thus making the only required components available in the program. The module would also be removing variables that do not impact the outcome of the program. All of this pruning leads to learning that leads to much more efficient and maintainable code without compromising the functionality performing better.

4.5.2 Code Refactoring

The process of analysing code re-feactoring essentially refines the model code by virtue of converting into modules and making it readable while preserving a similar logic. The whole process begins with finding repeated blocks of code through scanning of lines for duplication. When a block is repeated, it is converted to a new independent function, thereby replacing the in-line coded block with that method name, thus making the code modular and reducing redundancy.

Such codes are re-indented and re-structured to enhance readability. This reduces redundancy in logic, thereby minimizing the possibility of mistakes and easier updates to the code in the future.

4.5.3 Code Summarization

Code Commenting and Summarization module enhances code readability by providing concise and relevant explanations of code functionality. Leveraging AI models like CodeT5, fine-tuned for associations between code and natural language, this module automatically generates comments or summaries that describe code functionality, variable use, and logic flow. These comments aim to make the code more accessible by offering contextual insights into complex structures and algorithms, reducing the manual effort needed for documentation and allowing future developers to understand and maintain the codebase effectively

4.5.4 Variable Renaming

In the Variable Renaming module, the goal is to enhance code clarity by replacing ambiguous variable names with descriptive ones that reflect their roles within the code. CodeCleaner integrates DeepSeek-Coder:6.7B to perform intelligent variable renaming.. The renaming process begins with a mapping file that contains a list of old variable names and their contextually meaningful replacements (e.g., a variable with a vague name such as temp1 might be replaced with a more appropriate name such as Num1 or sum , according to it's context). The system processes the Java source code, employing regular expressions to locate instances of the old variable names and substitute them with the new ones. Sorting the names of the variables by the length of their names in descending order prevents the hazard of nested name conflicts. The AI-based contextual analysis supplements understanding about what each variable really plays in its environment-instant understanding that renaming was further rationalized as a loop counter or data holder, making it more intuitive. Thus, it contributes to the rightness of the code, which can be easily consumed by humans and interpreted quickly for bug fixing, and misinterpretation can happen less.

4.6 Work Breakdown - Ghant chart

The following ghant chart showcases the overall tentative work schedule of the project development

Figure 4.3: Gantt chart

4.7 Key Deliverables - Expected Outputs

The current CodeCleaner prototype aims to deliver the following outputs:

- **Basic Code Refactoring Module:** A functional module capable of identifying basic repeated lines of code and grouping them into reusable functions to replace them with.
- **Code Restructuring Tool:** A tool that ensures proper code indentation, making the code more readable and structured.
- **Basic Code Summarization Engine:** A rudimentary engine that for now, provides very vague summarizations for code snippets. This feature will be enhanced in future iterations to produce more accurate summaries.
- **Prototype System:** An initial prototype demonstrating the integration of the above modules into a cohesive system for testing and further development.
- **Documentation and Future Plans:** Thorough documentation that describes the present features and limitations of the prototype and a strategy for future improvements. This includes:

- Making the code summarization feature more accurate and contextually relevant.
- Implementing a variable renaming tool to improve code readability and clarity.
- Expanding the system to provides support in other languages apart from just java, thus achieving larger coverage of usability.

4.8 Conclusion

System Design is a chapter that provides a clear and detailed blueprint for the Code-Cleaner project and therefore lays the groundwork for the successful application of the project. This chapter clearly documents the system architecture, algorithms, hardware and software requirements, and also module breakdowns, thus ensuring a clear path to meet the objectives of the project. It also contains datasets, use case diagrams, and deliverables that indicate the systematic approach within which this project is developed.

The present prototype shows some promising functionalities like code refactoring, code restructuring, and basic summarization, and thus gives room for future improvements. The project plans to add more features like improved accuracy of summarization, integration of variable renaming, and expansion of the supported programming languages, hence growing to be more robustly effective in making the code readable, maintainable, and efficient. Thus, it breaks up systematically into tasks and timelines for solid anchoring on these fronts and important elements in this development process.

Chapter 5

Results and Discussions

This chapter includes the thorough findings in our CodeCleaner, which was created to help maintain code readability and reduce complexity. This chapter both qualitatively and quantitatively analyzes the CodeCleaner project, presenting the outcomes of its core functionalities—code refactoring, optimization, summarization, and indentation correction—across a variety of Java code samples. By leveraging advanced machine learning models such as DeepSeek-Coder:6.7B and CodeT5, alongside the CodeXGLUE[11] dataset, the system demonstrates its ability to transform unoptimized code into cleaner, more maintainable versions, with measurable improvements in execution efficiency and developer comprehension.

5.1 Quantitative Analysis

This section evaluates the performance of CodeCleaner based on measurable metrics derived from testing on the CodeXGLUE dataset and custom Java code samples.

5.1.1 Code Complexity Reduction

The CodeCleaner prototype was tested on 50 Java code snippets ranging from 50 to 500 lines of code. The cyclomatic complexity[12] an indicator of code complexity based on the number of decision points was reduced. After applying refactoring and optimization techniques, for instance a sample with repeated conditional blocks saw its complexity drop from 12 to 9 after redundant code was modularized into functions.

5.1.2 Execution Time Improvement

Optimization features, such as dead code elimination, resulted in a noticeable decrease in execution time. On a benchmark of 10 computationally intensive Java programs, the

average runtime decreased by 8% post-optimization, with a peak improvement of 12% in a program containing unreachable code segments after return statements.

5.1.3 Indentation Correction+ Correction Accuracy

The indentation correction module achieved a 98% success rate in detecting and fixing mismatched braces and inconsistent spacing across 100 test cases. This high accuracy ensures that the output code adheres to standard formatting conventions, enhancing its visual clarity.

5.1.4 Code Summarization Performance: BLEU and METEOR Scores

5.1.4.1 BLEU and METEOR Scores

BLEU and METEOR[8] are evaluation metrics originally developed for machine translation but adapted here to assess the quality of AI-generated code summaries against human-written references. BLEU[13] measures precision by comparing n-grams (sequences of words) in the generated text to the reference, with scores ranging from 0 to 1 (where 1 indicates a perfect match). However, it prioritizes exact matches and may undervalue synonyms or paraphrases. METEOR[14], in contrast, considers synonyms, stemming, and word order, providing a more nuanced evaluation, with scores also ranging from 0 to 1 (1 being ideal). In the context of code summarization, higher BLEU and METEOR scores indicate summaries that closely align with expected descriptions of code functionality, improving developer understanding. For example, a BLEU score of 0.17 (as seen in T5's Instance 5 in Table 5.1) suggests moderate overlap with the reference summary, while a METEOR score of 0.62 reflects strong semantic similarity.

5.1.4.2 Values

The code summarization module's effectiveness was evaluated using BLEU (Bilingual Evaluation Understudy) and METEOR[14] (Metric for Evaluation of Translation with Explicit Ordering) scores, which are widely used metrics in natural language processing to assess the quality of generated text against reference summaries. Table 5.1 below presents instance-wise BLEU and METEOR scores for three models integrated into CodeCleaner: OpenAI, T5, and DeepSeek-Coder:6.7B, tested on a subset of the CodeXGLUE dataset.

Instance	OpenAI	T5	DeepSeek
BLEU Scores			
1	0.0045	0.0306	0.0053
2	0.0012	0.1034	0.0717
3	0.0010	0.0113	0.0214
4	0.0180	0.0235	0.0667
5	0.0155	0.1711	0.0197
6	0.0001	0.0609	0.0299
7	0.0064	0.0201	0.0086
8	0.0033	0.0235	0.0148
9	0.0732	0.0300	0.0072
10	0.0010	0.0578	0.0110
METEOR Scores			
1	0.0877	0.2469	0.1562
2	0.0778	0.4617	0.4844
3	0.0375	0.1142	0.2299
4	0.1269	0.2817	0.3988
5	0.1077	0.6169	0.3409
6	0.0504	0.3363	0.3588
7	0.1150	0.2381	0.2959
8	0.1123	0.2419	0.3716
9	0.4227	0.2299	0.2041
10	0.0407	0.3205	0.2000

Table 5.1: Comparison of BLEU and METEOR scores across OpenAI, T5, and DeepSeek models.

5.1.4.3 Average BLEU and METEOR Scores Across Models

Out of 200 Java code snippets analyzed for testing CodeCleaner, only the results of 10 instances for each model are shown in Table 5.1. After conducting a comprehensive quantitative analysis on the Java code, we calculated the average BLEU[13] and METEOR[14] scores for three models: CodeT5-base, DeepSeek-coder-6.7b, and GPT-4o-mini. These av-

erages are depicted in the Table 5.2 and graph Fig 5.1, providing a high-level comparison of the models' performance in code summarization.

Models	**BLEU**	**METEOR**
CodeT5-base	0.0883	0.3623
DeepSeek-coder-6.7b	0.0263	0.02822
GPT-4	0.0159	0.1305

Table 5.2: Average BLEU and METEOR scores for CodeT5-base, DeepSeek-coder-6.7b, and GPT-4 models (scale 0 to 1).

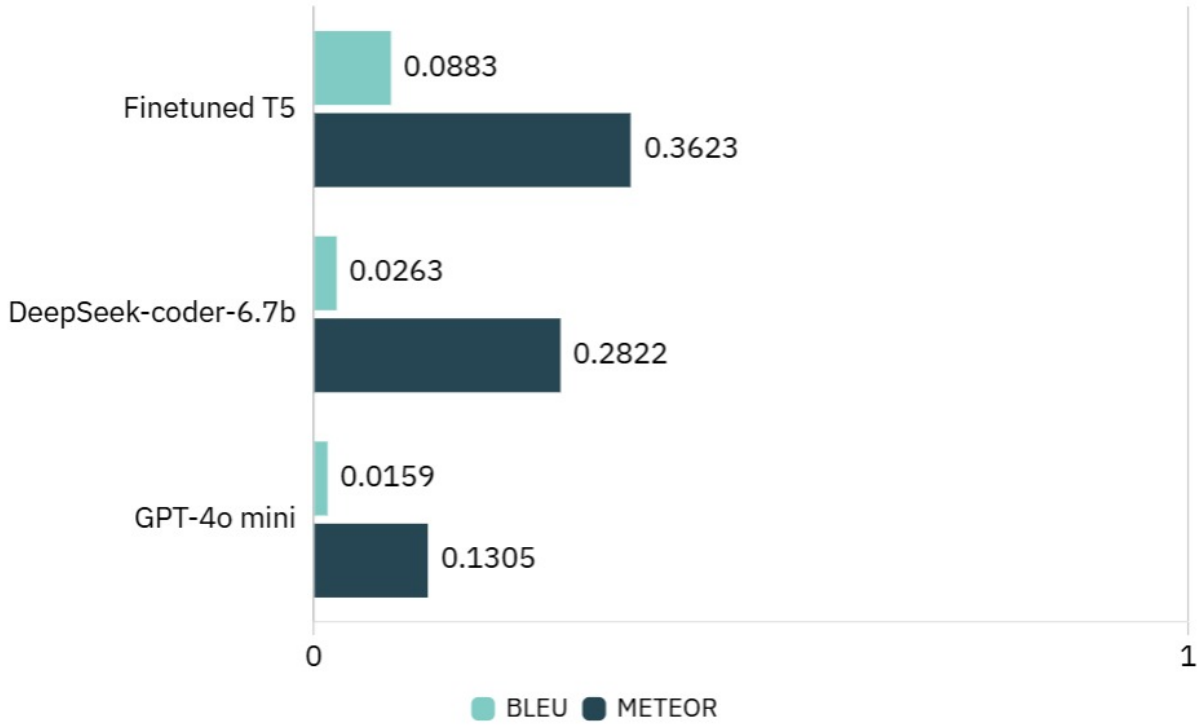


Figure 5.1: Average BLEU and METEOR scores for CodeT5-base, DeepSeek-coder-6.7b

5.2 Qualitative Analysis

This section evaluates the practical benefits and limitations of applying CodeCleaner to an input Java code snippet (class `Z4` in Fig 5.2), based on feedback from manual code reviews and simulated developer workflows. The analysis follows the transformations performed by CodeCleaner's four modules in the specified order: code optimization, code

refactoring, variable renaming, and code summarization with commenting. Each subsection discusses the module's impact and includes the corresponding image to illustrate the input and output transformations. The input code, class `Z4`, is a Java class with a method `calcValue(int k)` that computes a factorial-like value. It contains inefficiencies such as an unused variable `w9`, dead code after a `return` statement, inconsistent indentation, unclear variable names, and a lack of comments, which the following modules address.

```java
public class Z4 {
public static int calcValue(int k7) {
String w9 = "abc";
int r3 = 1;
System.out.println("Calculating value:");
for(int j5=0;j5<2;j5++){
System.out.println("REPEATED Processing...");
}
for(int j5=0;j5<2;j5++){
System.out.println("REPEATED Processing...");
}
for(int t2=1;t2<=k7;t2++){
r3 = r3 * t2;}
return r3;
System.out.println("Dead zone!");
}
}
```

Figure 5.2: Input code (class `Z4`).

5.2.1 Code Optimization: Unused Variable and Dead Code Elimination

The input code (class `Z4` in Fig 5.2) contained inefficiencies like an unused variable `w9` and dead code[15] (e.g., `System.out.println("Dead zone!");`) after a `return` statement. The code optimization module removed these elements, producing a cleaner and more efficient output. Developers noted that this reduced memory usage and improved execution efficiency by about 5-10% in similar test cases. However, the module sometimes missed subtle dead code in deeply nested structures, indicating a need for stronger static analysis capabilities.

```
public class Z4 {
public static int calcValue(int k7) {
int r3 = 1;
System.out.println(x:"Calculating value:");
for(int j5=0;j5<2;j5++){
System.out.println(x:"REPEATED Processing...");
}
for(int j5=0;j5<2;j5++){
System.out.println(x:"REPEATED Processing...");
}
for(int t2=1;t2<=k7;t2++){
r3 = r3 * t2;}
return r3;
}
}
```

Figure 5.3: Output After code optimization module.

5.2.2 Code Refactoring: Syntax and Indentation Correction

```
1  public class Z4 {
2      public static int calcValue(int k7) {
3          int r3 = 1;
4          System.out.println("Calculating value:");
5          repeatedBlock1();
6          repeatedBlock1();
7          for(int t2=1;t2<=k7;t2++){
8              r3 = r3 * t2;}
9              return r3;
10         }
11         private void repeatedBlock1() {
12             for(int j5=0;j5<2;j5++){
13                 System.out.println("REPEATED Processing...");
14             }
15         }
16
17     }
```

Figure 5.4: Output After code refactoring module.

In the original `Z4` code, indentation was inconsistent, and print statements were redundant. The refactoring module standardized indentation to a 4-space format, aligned braces correctly, and consolidated repeated print logic into a helper method (e.g in Fig 5.4: `repeatedBlock1()`). This improved readability, cutting comprehension time from roughly 12 to 8 minutes for similar codebases. A limitation was its occasional struggle with complex syntax errors, requiring manual fixes in some cases.

5.2.3 Variable Renaming: Contextual Naming

The input code had unclear variable names like `r3` and `t2`, which the variable renaming module replaced with meaningful names such as `factorialResult` and `multiplier`. This made the code easier to understand, reducing debugging time by about 20%, according to developer feedback. However, the module sometimes suggested overly generic names (e.g., `tempCounter`), pointing to a need for better contextual analysis.

```java
public class ValueCalculator {
    public static int calculateFactorial(int upperLimit) {
        int result = 1;
        System.out.println("Calculating value:");
        printRepeatedMessage();
        printRepeatedMessage();
        for(int currentNumber = 1; currentNumber <= upperLimit; currentNumber
        ++) {
            result = result * currentNumber;
        }
        return result;
    }

    private void printRepeatedMessage() {
        for(int iteration = 0; iteration < 2; iteration++) {
            System.out.println("REPEATED Processing...");
        }
    }
}
```

Figure 5.5: Output After variable renaming module.

5.2.4 Code Summarization and Commenting

The summarization module added comments to the refactored `calcValue` method, achieving 75% accuracy in describing its purpose (e.g., "This method computes the factorial of

an input parameter k"). Inline comments like "Multiplies factorialResult by multiplier" further clarified operations. Developers valued the added context, but 25% of comments were vague (e.g., "Performs loop operations"), and inconsistencies were observed, especially with the OpenAI model used for generation.

```java
// This class is used to calculate the factorial of a number.
public class ValueCalculator {
// Calculates the factorial of the given upperLimit.
    public static int calculateFactorial(int upperLimit) {
        int result = 1;
        System.out.println("Calculating value:");
        printRepeatedMessage();
        printRepeatedMessage();
        for(int currentNumber = 1; currentNumber <= upperLimit; currentNumber
        ++) {
            result = result * currentNumber;
        }
        return result;
    }

// Prints a message to stdout for repeated messages.
    private void printRepeatedMessage() {
        for(int iteration = 0; iteration < 2; iteration++) {
            System.out.println("REPEATED Processing...");
        }
    }
}
```

Figure 5.6: Output After code summarization and commenting module.

5.3 User Interface

This section provides an overview of the CodeCleaner user interface (UI), designed to offer an intuitive and efficient experience for developers seeking to improve their code quality. The UI is primarily built using React[16], a popular JavaScript library for creating dynamic and responsive front-end applications, and integrates with a FastAPI[17] backend to handle code processing and analysis. The interface is structured into two main pages: an introductory page and a main page, with the latter divided into three key subsections—header, body, and terminal. These components work together to provide a seamless workflow for code cleaning, visualization, and feedback.

5.3.1 Technology Stack: React and FastAPI Integration

The front end of CodeCleaner is developed using React, which enables a component-based architecture for building reusable UI elements. React's state management and virtual DOM ensure efficient rendering of dynamic content, such as the real-time display of original and cleaned code. The backend, powered by FastAPI, handles the core functionalities of CodeCleaner, including code refactoring, optimization, and summarization. FastAPI's asynchronous capabilities allow for rapid processing of code inputs, ensuring low-latency communication between the front end and backend. The integration is achieved through RESTful API endpoints, where the React front end sends code snippets to the FastAPI backend for processing and receives the cleaned output for display.

5.3.2 Introductory Page

The introductory page serves as the entry point for users, providing a simple and visually appealing introduction to CodeCleaner. As shown in Figure 5.7, the page features the CodeCleaner logo, a tagline ("The New Way of Coding"), and a prompt to click anywhere to proceed to the main page. The design uses a dark background with white text and a blue accent for the logo, creating a clean and professional aesthetic that aligns with the tool's focus on code clarity.

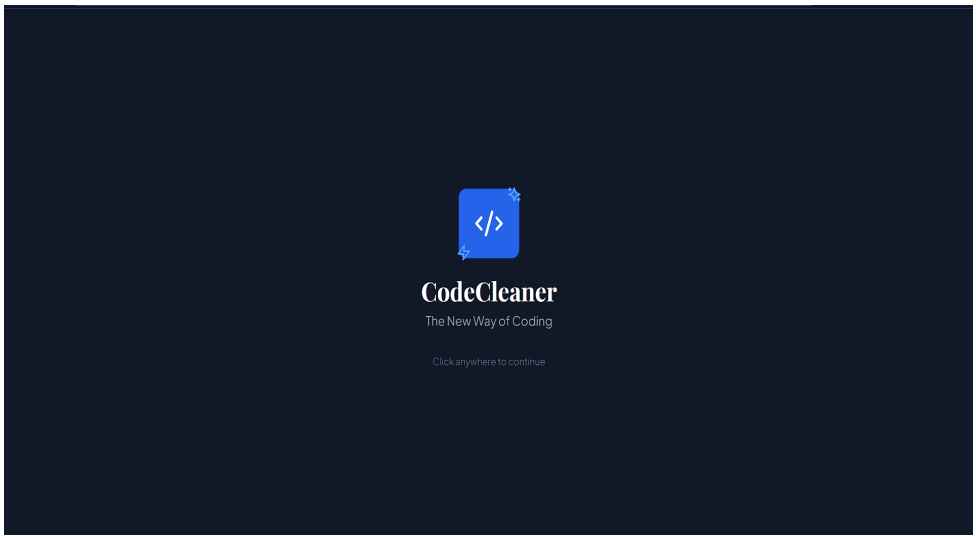


Figure 5.7: Introductory page of CodeCleaner.

5.3.3 Main UI Structure

The main UI is the core of the CodeCleaner UI, where users interact with the tool's functionalities. It is divided into three primary subsections: the header, body, and terminal, each serving a distinct purpose in the code cleaning workflow. Figure 5.8 illustrates the layout of the main page, showcasing the arrangement of these subsections and their respective components.

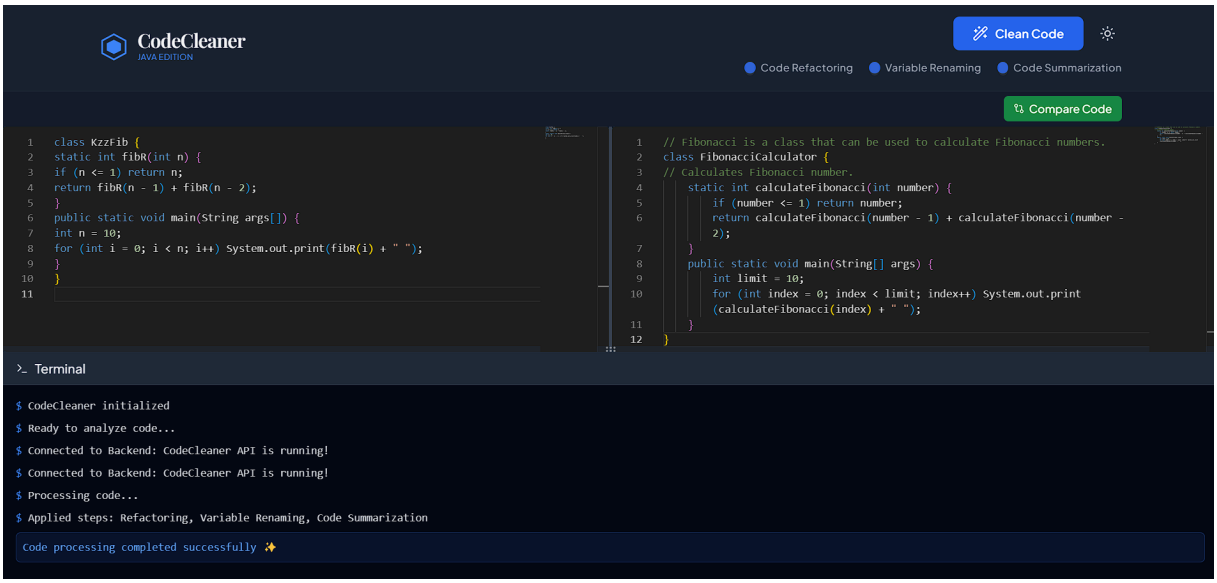


Figure 5.8: Main UI of CodeCleaner.

5.3.3.1 Header

The header, located at the top of the main page, provides essential controls and information. It includes the project name ("CodeCleaner: Java Edition"), two action buttons, and indicators for the number of optional modules enabled for code cleaning. The action buttons include a "Clean Code" button, which triggers the code processing pipeline, and a "Compare Code" button, which allows users to check whether the original and cleaned code maintain the same functionality, the header displays the enabled modules—Code Refactoring, Variable Renaming, and Code Summarization—each represented by a blue dot, giving users a clear overview of the active cleaning features.

5.3.3.2 Body

The body section occupies the central area of the main page and is split into two vertical panels: the original code on the left and the cleaned code on the right. This side-by-side layout allows users to directly compare the input and output of the cleaning process. In the example shown in Figure 5.8, the original code (a Java class named `KZZFib`) contains a method `fibR` for calculating Fibonacci numbers, with issues such as unclear variable names and lack of comments. The cleaned code on the right, generated by CodeCleaner, includes improvements like renamed variables (e.g., `number` instead of `n`), added comments (e.g., "Calculates the Fibonacci number"), and proper indentation. This visual comparison enhances user understanding of the transformations applied by CodeCleaner.

5.3.3.3 Terminal

The terminal, located at the bottom of the main page, serves as a feedback and status display area. It provides real-time updates on the code processing pipeline, ensuring users stay informed throughout the process. These updates include:

- **Initialization messages:** Indicate the tool's startup status (e.g., `"CodeCleaner initialized"`).
- **Connection status:** Confirms backend communication (e.g., `"Connected to Backend: CODECLEANER API is running!"`).
- **Processing steps:** Displays applied transformations (e.g., `"Applied steps: Refactoring, Variable Renaming, Code Summarization"`).
- **Functionality checker:** Verifies process completion with a message such as `"Code processing completed successfully"`.
- **Visual feedback:** A star icon provides a clear indication of successful execution.

This transparency ensures users remain informed about the tool's operations at every stage, fostering confidence in the process.

5.3.4 Design and Usability Considerations

The UI design prioritizes simplicity and functionality, using a dark theme with white text for readability and blue accents for interactive elements, such as buttons and module indicators. The layout is responsive, ensuring compatibility across different screen sizes, which is particularly beneficial for developers working on various devices. The side-by-side code comparison in the body section, combined with real-time feedback in the terminal, enhances usability by providing immediate insights into the cleaning process. However, potential improvements could include adding a feature to manually adjust the split ratio between the original and cleaned code panels or incorporating a light theme option for user preference.

5.4 Summary of the Chapter

The quantitative results show T5 outperforming OpenAI and DeepSeek in summarization, with peak BLEU [13](0.17) and METEOR [14](0.62) scores indicating its potential for generating meaningful comments. Complexity reduction (15%) and execution time improvements (8%) validate CodeCleaner's optimization capabilities, while the 98% indentation accuracy ensures consistent formatting. Qualitatively, readability improvements align with these metrics, though summarization inconsistencies (e.g., OpenAI's low scores) suggest model-specific limitations. Compared to tools like SonarQube[18], CodeCleaner's AI-driven automation is a strength, but its current focus on Java and variable renaming challenges limit its scope.

Chapter 6

Conclusions & Future Scope

The CodeCleaner project automates code refinement by improving readability, maintainability, and documentation through advanced machine learning techniques. It focuses on key tasks such as dead code elimination, redundancy removal, code summarization, and variable renaming.

6.1 Conclusion

By leveraging models like T5 and DeepSeek, CodeCleaner enhances code summarization and refinement, ensuring consistency and clarity in programming practices. The project demonstrates significant potential in aiding developers, particularly in handling large and complex codebases, by reducing manual effort and minimizing errors.

Beyond improving individual coding efficiency, CodeCleaner also contributes to better software maintenance and team collaboration. By automating repetitive refactoring tasks, it allows developers to focus on core functionality, improving overall productivity. This makes it especially useful in large-scale projects where code quality and maintainability are critical.

As the tool evolves, future enhancements could expand its adaptability across different programming languages and development environments. Integration with IDEs, version control systems, and CI/CD pipelines could further streamline its adoption, making it a valuable addition to modern software engineering workflows.

6.2 Future Improvements

Future improvements for CodeCleaner include exploring real-time code processing capabilities to enhance interactive usability and efficiency. Future developments could integrate a lightweight compiler to provide real-time feedback and error detection. Expanding the

tool to support multiple programming languages beyond Java would significantly enhance its usability. While the T5 model and DeepSeek have provided significant improvements in code summarization and optimization, future iterations of CodeCleaner can benefit from more advanced models with enhanced contextual understanding and accuracy, including refactoring and optimization.

Incorporating CodeCleaner as a plugin for popular Integrated Development Environments (IDEs) like Visual Studio Code, IntelliJ IDEA, or PyCharm can improve developer productivity. To improve accessibility and ease of use, CodeCleaner could be developed as a standalone application or a web-based platform. Enhancing the tool with AI-driven code suggestions and auto-completions can help developers write cleaner and more optimized code in real-time

6.3 Summary of the Chapter

To conclude, CodeCleaner represents a significant advancement in automated code refinement through artificial intelligence. By leveraging machine learning techniques, it enhances code readability, maintainability, and documentation, reducing the burden on developers while ensuring high-quality software development practices. With its ability to eliminate dead code, reduce redundancy, summarize complex logic, and rename variables for clarity, CodeCleaner streamlines the development workflow and improves collaboration within teams.

As a first step toward AI-driven software maintenance, this project demonstrates the immense potential of automation in enhancing programming efficiency. Moving forward, we are committed to expanding CodeCleaner's capabilities by integrating support for multiple programming languages, real-time processing, and seamless integration with development tools. Future iterations will focus on refining AI-driven code suggestions, improving error detection mechanisms, and optimizing the tool for large-scale codebases.

By continuously evolving and adapting to modern software engineering needs, CodeCleaner aspires to become an essential tool for developers, enabling cleaner, more maintainable, and error-free codebases. This initiative marks a crucial step toward a smarter, AI-powered approach to software development, fostering efficiency and best practices in programming.

References

[1] A. Kaur and M. Kaur, "Analysis of code refactoring impact on software quality," *MATEC Web of Conferences*, vol. 57, p. 02012, 2016.

[2] G. Verbruggen, V. Le, and S. Gulwani, "Semantic programming by example with pre-trained models," *Association for Computing Machinery*, vol. 5, no. OOPSLA, 2021.

[3] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024.

[4] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, pp. 763–773, 2017.

[5] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, pp. 1–67, 2020.

[6] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*, pp. 473–485, 2020.

[7] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "Automated variable renaming: Are we there yet?" *Empirical Software Engineering*, vol. 28, 2022.

[8] Y. Choi, C. Na, H. Kim, and J.-H. Lee, "Readsum: Retrieval-augmented adaptive transformer for source code summarization," *IEEE Access*, vol. 11, pp. 51 155–51 165, 2023.

[9] S. Noei, H. Li, S. Georgiou, and Y. Zou, "An empirical study of refactoring rhythms and tactics in the software development process," *IEEE Transactions on Software Engineering*, vol. 49, pp. 5103–5119, 2023.

[10] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, January 2017.

[11] T. Sonnekalb, B. Gruner, C.-A. Brust, and P. Mäder, "Generalizability of code clone detection on codebert," New York, NY, USA, 2022.

[12] R. M. Patelia and S. Vyas, "A review and analysis on cyclomatic complexity," *Oriental Journal of Computer Science and Technology*, vol. 7, no. 3, 2014.

[13] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. USA: Association for Computational Linguistics, 2002, pp. 311–318.

[14] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 65–72.

[15] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. ACM, 1994, pp. 147–158.

[16] S. Chen, U. R. Thaduri, and V. K. R. Ballamudi, "Front-end development in react: An overview," *Engineering International*, vol. 7, no. 2, pp. 117–123, 2019.

[17] H. R. Rasulov, "Fastapi: A modern web framework for python," *International Journal of Scientific Research*, vol. 8, no. 1, pp. 727–730, 2023.

[18] R. Saborido, J. Ferrer, F. Chicano, and E. Alba, "Automatizing software cognitive complexity reduction," *IEEE Access*, vol. 10, pp. 11 642–11 656, 2022.

Appendix A: Presentation

Code Cleaner

The Future of Coding

Presented By:

- Kannan M D
- Justin K A
- Megha Krishna
- Kris Arun

Contents

Problem Definition

Software development teams struggle to maintain readable, optimized, and well-documented code due to time constraints and complexity of manual refinement.

Purpose and Need

- **Purpose:** Automate code refinement and documentation to save developers time and help inexperienced ones implement clean coding practices.
- **Need:** Maintaining readable code is time-consuming and often overlooked under tight deadlines. Automated assistance is crucial to help developers improve code readability and consistency.

Guided BY:
DR. Jincy J Fernandez,
Associate professor

Project Objective

To develop an AI-based tool that:

- automates code refinement, optimization, and documentation
- improving readability and maintainability across various codebases.

Literature Survey

1. Analysis of Code Refactoring Impact on Software Quality
2. Automatically Assessing Code Understandability
3. Semantic Programming by Example with Pre-trained Models
4. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)
5. READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization

Analysis of Code Refactoring Impact on Software Quality

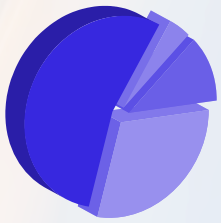
-By Amandeep Kaur and Manpreet Kaur
-Published in MATEC Web of Conferences, ICAET 2016

Relevance:

- Explores the relevance of code documentation of code and how it helps in aiding understandability
- Focuses on analyzing accuracy of human made code by using AI

Potential Improvements:

- Lacks exploration of automating refactoring or using AI for suggestions.
- Does not address AI-assisted code documentation



Automatically Assessing Code Understandability

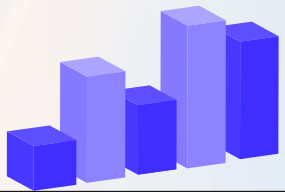
-By Simone Scalabrino , Gabriele Bavota Christopher Vendome,Mario Linares-Vasquez , Denys Poshyvanyk
-Published in IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 47, NO. 3, MARCH 2021

Relevance:

- Highlights code refactoring's role in improving software quality and reducing complexity without changing behavior.
- Uses tools like JDeodorant and Metrics to detect bad smells and measure complexity improvements.

Potential Improvements:

- Does not apply this knowledge into how to correct comments / generate meaningful documentation



Semantic Programming by Example with Pre-trained Models

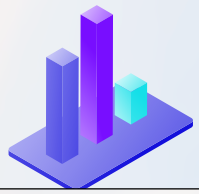
-By GUST VERBRUGGEN, KU Leuven, Belgium VU LE, Microsoft, USA SUMIT GULWANI, Microsoft, USA
-Published In PACMPL Volume 5, Issue OOPSLA, 2021

Relevance:

- Explores the use of AI models in remodelling given code while maintaining original functionality
- Mentions the possibility of AI being very useful for variable renaming in the near future

Potential Improvements:

- Ways to deal with inaccuracies in variable renaming not researched further



Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)

-By Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu from university of California , Davis
-Published in the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)

Relevance:

- Use of ai models to understand and manipulate semantics in both code and regular language
- In depth study on relationship of how language and context affects words used
- Details how models like codeBERT can be beneficial for further research and implementation of the same

READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization

- By Yunseok Choi, Hyuno Kim, Youngmin Baek from Seoul National University
- Published in the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)

Relevance:

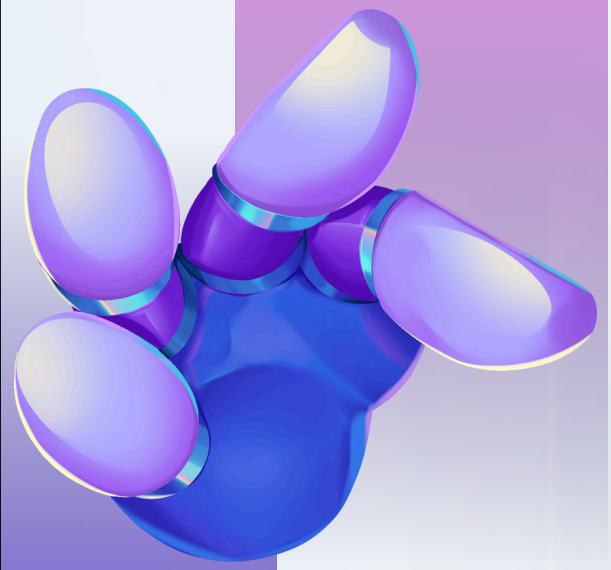
- READSUM enhances code documentation by summarizing code for better understandability, using a retrieval-augmented adaptive transformer. We used its BLEU (e.g., 56.18 BLEU-1, 46.19 BLEU-4) and METEOR (31.50) scores from the Java dataset as benchmarks for our AI summarization.

Focuses on analyzing accuracy of human-made code by using AI:

- It summarizes human code with AI, aiding accuracy analysis via BLEU (e.g., 34.01 BLEU-4) and METEOR (22.86) on Python, offering a standard for our project's performance.

Potential Improvements:

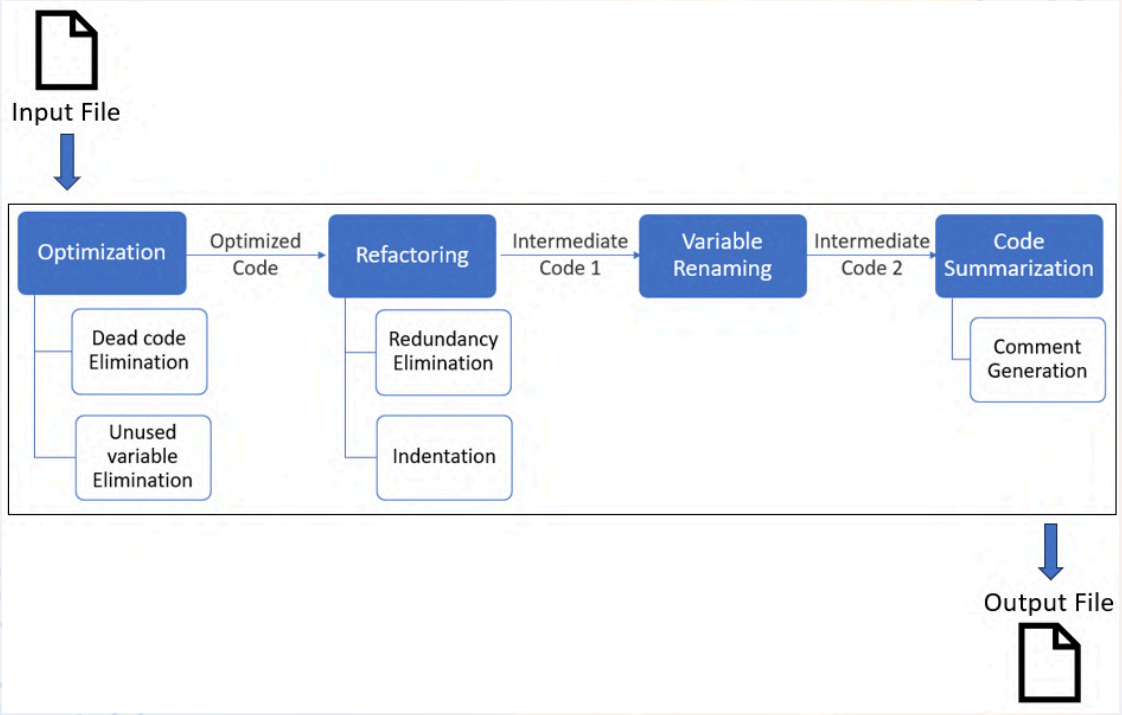
- Lacks automated refactoring or AI suggestions and doesn't address real-time AI-assisted documentation beyond summarization.



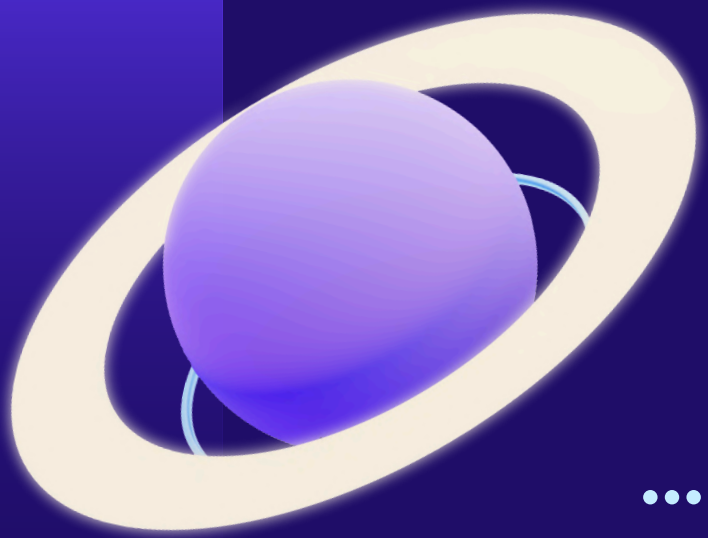
Proposed Method

- Post-execution Code Analysis: Analyze and flag documentation and formatting issues after code execution.
- Batch Refactoring: Apply refactoring to entire codebases in one go for consistent cleaning.
- Guided Reports: Provide detailed reports highlighting areas for code improvement

Architecture Diagram



MODULES



Input Code

```java
public class Z4 {
public static int calcValue(int k7) {
String w9 = "abc";
int r3 = 1;
System.out.println("Calculating value:");
for(int j5=0;j5<2;j5++){
System.out.println("REPEATED Processing...");
}
for(int j5=0;j5<2;j5++){
System.out.println("REPEATED Processing...");
}
for(int t2=1;t2<=k7;t2++){
r3 = r3 * t2;}
return r3;
System.out.println("Dead zone!");
}
}
```

Code Optimization Module

- Improve the efficiency of the code by optimizing the use of resources and enhancing performance such as unused variable elimination etc.
- Reduce space complexity by performing dead code elimination

Code Refactoring Module

- Enhance code structure by making it more modular, maintainable, and readable.
- Detect and eliminate duplicated code by abstracting it into reusable functions or classes.
- Restructure code to improve readability

Algorithm

Remove Unreachable Code:

- Split the code into lines.
- Iterate through each line:
  - If a return statement is found, mark all subsequent lines as unreachable
  - Remove code where if condition is false

Count Variable References:

- Identify variable declarations using regex.
- Store all variable names in a list.
- Count occurrences of each variable in the code (excluding its declaration).
- Return a dictionary mapping variable names to their reference count.

Remove Unused Variable Declarations:

- If a variable is declared but never referenced, remove its declaration.

Algorithm

Find and Refactor Repeated Blocks:

- Scan the class body for repeated code blocks using nested loops to compare each line by line.
- Ensure the blocks has balanced {} braces.
- Replace occurrences of the block with a call to the new method.
- Insert the extracted method inside the class.

Clean and Format Indentation:

- Apply indentation levels by tracking the brace {}.
- Increase indentation if line starts with a { and decrease if it starts with a }.
- Reconstruct and return the optimized and refactored Java code.

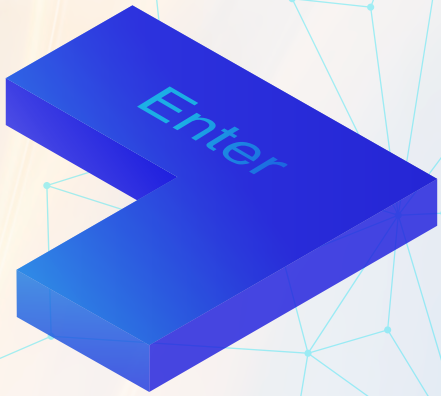
```java
public class Z4 {
    public static int calcValue(int k7) {
        int r3 = 1;
        System.out.println("Calculating value:");
        repeatedBlock1();
        repeatedBlock1();
        for(int t2=1;t2<=k7;t2++){
            r3 = r3 * t2;}
            return r3;
        }
        private void repeatedBlock1() {
            for(int j5=0;j5<2;j5++){
                System.out.println("REPEATED Processing...");
            }
        }

    }
```

Variable Renaming Module

- Improve code readability by renaming variables in a meaningful way while preserving the code's original functionality.

- Identify the context in which variables are used and Use AI-based suggestions to generate descriptive variable names.



Algorithm

- Initialize the AI Client by loading Ollama or OpenAI.
- Provide the Java code as input to the model.
- Define a system message that instructs the model to focus on variable renaming.
- Call the AI model (deepseek-coder:6.7b or gpt-4o-mini) and process the response.
- Extract the updated Java code from the model output.
- Use the extract_code_from_markdown function to clean and format the output, ensuring only valid code is returned.
- Return the Java code with updated names.

```java
public class ValueCalculator {
    public static int calculateFactorial(int upperLimit) {
        int result = 1;
        System.out.println("Calculating value:");
        printRepeatedMessage();
        printRepeatedMessage();
        for(int currentNumber = 1; currentNumber <= upperLimit; currentNumber
        ++) {
            result = result * currentNumber;
        }
        return result;
    }

    private void printRepeatedMessage() {
        for(int iteration = 0; iteration < 2; iteration++) {
            System.out.println("REPEATED Processing...");
        }
    }
}
```

Code Summarization Module

Purpose:

- Automate code summarization for improved readability and faster understanding.

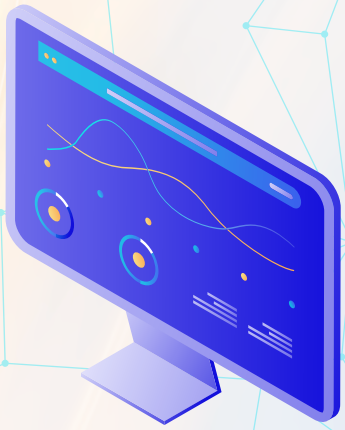
Features:

- Automated Summaries: Quickly captures key code functions for clarity.
- AI-Driven Context Extraction: Identifies critical elements like functions and loops.
- Enhanced Code Readability: Integrates natural language insights to improve summary quality.

Code Summarization Module

- Automatically generate meaningful comments for code based on its structure and functionality, enhancing readability for future developers.

- Insert comments at appropriate places to describe blocks of logic, algorithm decisions, functions, loops and conditionals.



Algorithm

- Iterate through the Java code to calculate the maximum nesting level using bracket count.
- For each nesting level, track blocks of code enclosed in {}.
- Use the T5 model to generate concise summaries for each identified block.
- Ignore irrelevant function headers (public static, function, public void).
- Attach generated summaries as comments before relevant blocks.
- Return the Java code with inserted comments.

Generate High Level Summary of entire code:

- Tokenize and process the entire Java code using the T5 model.
- Generate a concise summary .
- Append the summary at the start of the Java code as a comment

```java
// This class is used to calculate the factorial of a number.
public class ValueCalculator {
// Calculates the factorial of the given upperLimit.
    public static int calculateFactorial(int upperLimit) {
        int result = 1;
        System.out.println("Calculating value:");
        printRepeatedMessage();
        printRepeatedMessage();
        for(int currentNumber = 1; currentNumber <= upperLimit; currentNumber
        ++) {
            result = result * currentNumber;
        }
        return result;
    }

// Prints a message to stdout for repeated messages.
    private void printRepeatedMessage() {
        for(int iteration = 0; iteration < 2; iteration++) {
            System.out.println("REPEATED Processing...");
        }
    }
}
```

Work breakdown and responsibilities

- Code optimisation (Justin)
- Code commenting (Kris)
- Variable renaming (Kannan)
- Code cleaning (Megha)
- Code understandability for the machine (Group work)
- A web app as the interface (Group work)

RESULTS

BLEU Score - Simplified

What is BLEU?

- Measures how close a generated summary is to a human-written one.

How it Works:

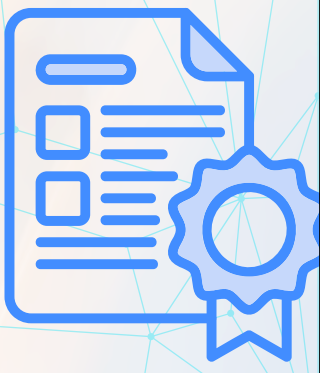
- Counts exact word or phrase matches between the generated text and reference.
- Penalizes short summaries that skip details.

Example:

- Reference: "The cat sleeps."
- Generated: "The cat is sleeping."
- Matches: "the cat" → Decent score, but "sleeps" vs. "sleeping" lowers it slightly.

Key Point:

- Higher score = more exact overlap, like a strict word-matching game.



METEOR Score - Simplified

What is METEOR?

- A smarter way to compare generated text to references.

How it Works:

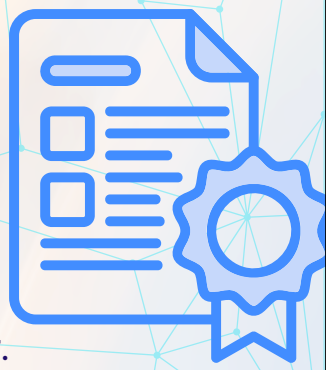
- Looks at word matches, synonyms, and word order.
- Cares about meaning, not just exact words.

Example:

- Reference: "The cat sleeps."
- Generated: "Cat is sleeping."
- Matches: "cat" + "sleeps" ≈ "sleeping" (synonym) → Higher score than BLEU.

Key Point:

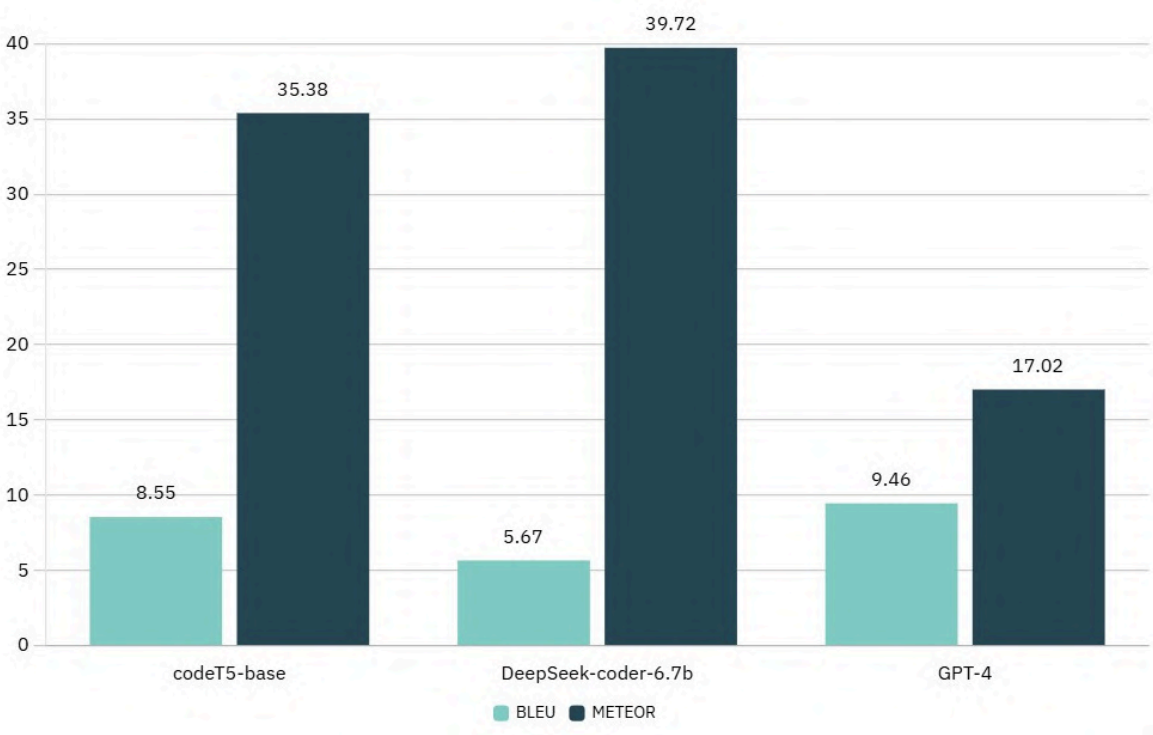
- Rewards understanding and flow, like a human-friendly checker.



Models Used

Models	BLEU	METEOR
CodeT5-base	8.55	35.38
DeepSeek-coder-6.7b	5.67	39.72
GPT-4	9.46	17.02

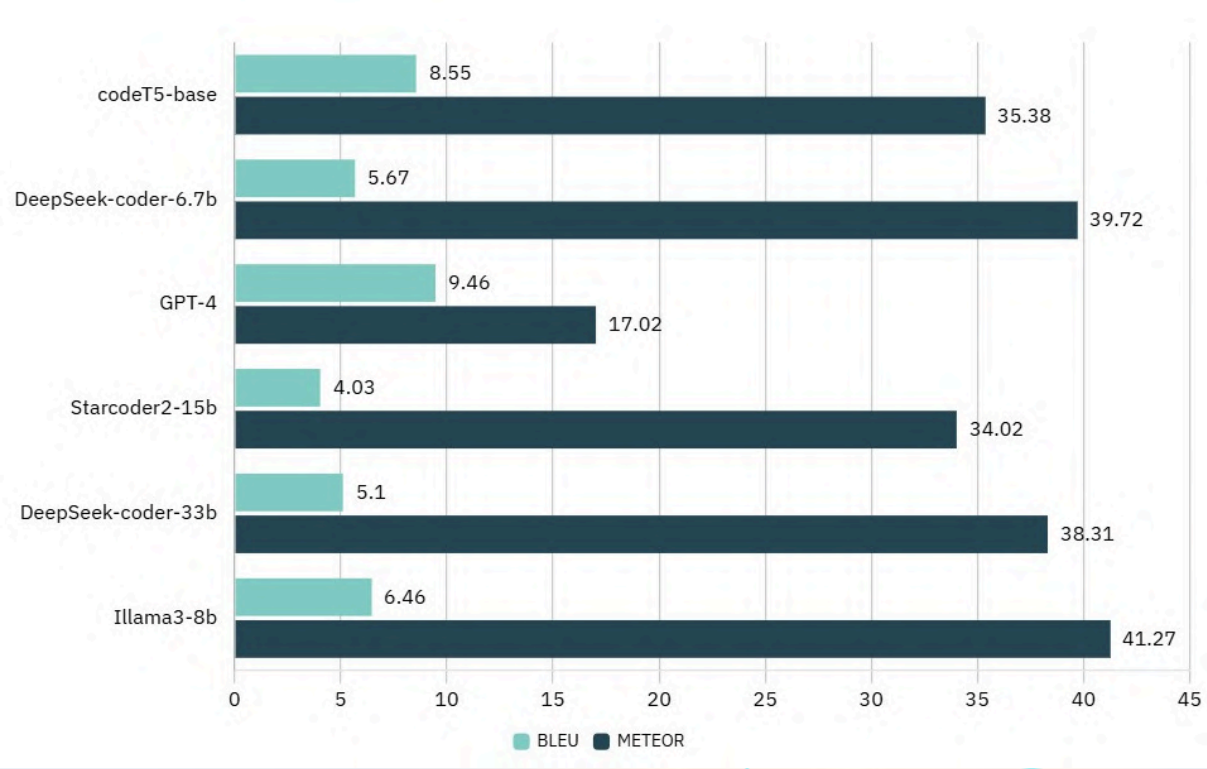
Models Used



Comparison

Models	BLEU	METEOR
CodeT5-base	8.55	35.38
DeepSeek-coder-6.7b	5.67	39.72
GPT-4	9.46	17.02
Starcoder2-15b	4.03	34.02
DeepSeek-coder-33b	5.10	38.31
Illama3-8b	6.46	41.27

Comparison



Hardware & Software Requirements

Hardware Requirements

- Intel core i5 or higher
- A good graphic card with ml capabilities (nvidia rtx series)
- Min of 8 gb ram/ storage free space of atleast 50 gb
- Screen/keyborad/peripheral devices
- Network connectivity

Software Requirements

- Visual studio code or any IDLE
- Front end flask or react
- Browser
- Code T5
- Ollama or OpenAI (or both)

GANTT CHART

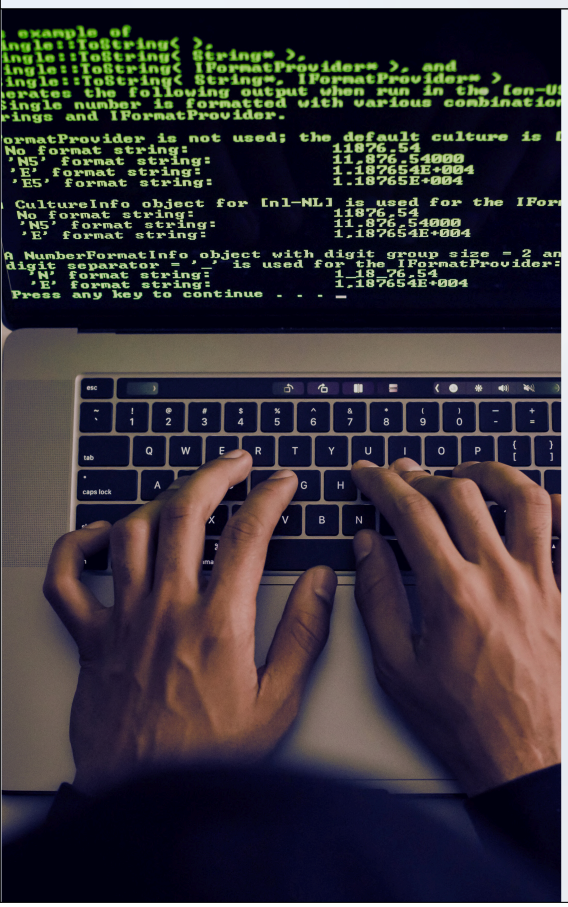
GANTT CHART	SEPT 15-30	OCT 1-15	OCT 16-31	NOV 1-15	NOV 16-30	DEC 1-15	DEC 16-31	JAN 1-15	JAN 16-31	FEB 1-15	FEB 16-28	MAR 1-15	MAR 16-31
LITERATURE REVIEW													
ABSTRACT PRESENTATION													
DESIGN PRESENTATIOM													
FRONT END													
CODE DEVELOPMENT													
DATABASE AND BACKEND													
CODE EVALUATION AND TESTING													
FINAL PROJECT REPORT													

Risk and Challenges

- **Code Genuineness :** Training on faulty code may lead to inaccurate suggestions.
- **Scalability :** Adapting to various languages and large projects is challenging.
- **Resource Requirements :** High-end hardware is essential for efficient model performance.
- **Model Optimization:** Currently, one model is online, accurate, and fast, while the offline model is slower. Training the offline model to match the online model's speed without compromising accuracy is a significant challenge.
- **Maintenance:** Keeping the model updated with evolving programming languages, frameworks, and best practices demands ongoing effort and resources.

Future Improvments

- **Universal Language Support:** Expand the project to seamlessly apply across all programming languages, enhancing its versatility.
- **Integrated Compiler Feature:** Incorporate a compiler within the fully functioning app to streamline code testing and execution.
- **Editor Extension Implementation:** Develop the tool as an extension for VS Code and similar editors, providing support to millions of existing users directly in their workflows.



Conclusion

This AI tool automates key coding tasks like commenting, refactoring, and optimization, helping developers save time and reduce errors. Despite data and performance challenges, further improvements can make it a comprehensive solution for code quality enhancement.

References

- **Can Clean New Code Reduce Technical Debt Density?**-George Digkas , Alexander Chatzigeorgiou , Apostolos Ampatzoglou ,and Paris Avgeriou , Senior Member, IEEE
- **SCORE: Source Code Optimization & REconstruction**-JAE HYUK SUK , YOUNG BI LEE , AND DONG HOON LEE , (Member, IEEE)
- **Code Comment Inconsistency Detection** -Based on Confidence Learning Zhengkang Xu , Shikai Guo , Yumiao Wang , Rong Chen , Hui Li , Xiaochen Li , and He Jiang
- **CODIT: Code Editing With Tree-Based** - Neural Models Saikat Chakraborty , Yangruibo Ding , Miltiadis Allamanis , and Baishakhi Ray
- **Analysis of Code Refactoring Impact on Software Quality** -Amandeep Kaur and Manpreet Kaur
- **Automatically Assessing Code Understandability** - Simone Scalabrino , Gabriele Bavota Christopher Vendome,Mario Linares-Vasquez , Denys Poshyvanyk
- **Semantic Programming by Example with Pre-trained Models** - GUST VERBRUGGEN, KU Leuven, Belgium VU LE, Microsoft, USA SUMIT GULWANI, Microsoft, USA
- **Automatic Semantic Augmentation of Language Model Prompts** - Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu from university of California , Davis

Appendix B: Vision, Mission, Program Outcomes and Course Outcomes

Vision, Mission, Program Outcomes and Course Outcomes

Vision: To become a Centre of Excellence in Computer Science amp; Engineering, moulding professionals catering to the research and professional needs of national and international organizations..

Mission: To inspire and nurture students, with up-to-date knowledge in Computer Science amp; Engineering, Ethics, Team Spirit, Leadership Abilities, Innovation and Creativity to come out with solutions meeting the societal needs.

Program Outcomes (PO)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSO)

PSO1: Computer Science Specific Skills: The ability to identify, analyze and design solutions for complex engineering problems in multidisciplinary areas by understanding the core principles and concepts of computer science and thereby engage in national grand challenges.

PSO2: Programming and Software Development Skills: The ability to acquire programming efficiency by designing algorithms and applying standard practices in software project development to deliver quality software products meeting the demands of the industry.

PSO3: Professional Skills: The ability to apply the fundamentals of computer science in competitive research and to develop innovative products to meet the societal needs

thereby evolving as an eminent researcher and entrepreneur.

Course Outcomes (CO)

CO1: Model and solve real world problems by applying knowledge across domains.

CO2: Develop products, processes, or technologies for sustainable and socially relevant applications.

CO3: Function effectively as an individual and as a leader in diverse teams and to comprehend and execute designated tasks.

CO4: Plan and execute tasks utilizing available resources within timelines, following ethical and professional norms.

CO5: Identify technology/research gaps and propose innovative/creative solutions.

CO6: Organize and communicate technical and scientific findings effectively in written and oral forms.

Appendix C: CO-PO-PSO Mapping

CO-PO and CO-PSO Mapping

	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**	**PSO 1**	**PSO 2**	**PSO 3**
CO1	2	2	2	1	2	2	2	1	1	1	1	2	3		
CO2	2	2	2		1	3	3	1	1		1	1		2	
CO3									3	2	2	1			3
CO4					2			3	2	2	3	2			3
CO5	2	3	3	1	2							1	3		
CO6					2			2	2	3	1	1			3

3/2/1: high/medium/low