Thesis Document for CodePure

Mohammad Hesham Elsayed, Mark Mounir Adly, Omar Hosny Badrawy, Zeyad Abdelnasser Elzayaty

Supervised by Dr. Sarah Nabil and Eng. Saja Saadoun

Submitted May 2025, in partial fulfillment of
the conditions for the award of the degree **BSc Computer Science.**

Faculty of Computer Science
Misr International University, Cairo, Egypt

Abstract

CodePure, is an extension for Visual Studio Code that aims to assist developers in writing cleaner, more maintainable code by identifying and addressing common code smells. By leveraging datasets and code metrics, the extension will provide real-time detection of code smells, helping developers refactor their code more effectively. In addition, the extension will feature a comprehensive code smell dashboard, offering a visual overview of detected issues. CodePure will also provide recommendations for common smells, making it easier for developers to implement changes immediately. With cross-language support, the extension will be useful to developers working in multiple programming languages.

Acknowledgements

We'd like to thank our supervisor, Dr.Sarah Nabil, as well as Engineer Saja Saadoun, for their patience, guidance, and support. Your broad knowledge and meticulous assistance was beneficial to us. We are grateful that you opted to assist us and that you continued to believe in us throughout the graduation year.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Academic

The issue of detecting code smells has been extensively studied in both academic and industrial contexts due to its critical importance in software maintenance and evolution. Code smells are indicators of potential design issues that, while not necessarily bugs, can lead to reduced software quality and increased technical debt. The term "code smell" was popularized by Fowler in 1999, referring to characteristics of code that may indicate deeper problems in the design, often requiring refactoring for long-term sustainability. Historically, researchers have focused on identifying and classifying different types of code smells, such as large classes, long methods, and duplicated code, all of which can negatively impact software understandability and maintainability.Santos et al. [2018]

This problem becomes especially interesting when we consider the complexity of modern software systems. As software evolves, technical debt accumulates due to poor design decisions, pressure to meet deadlines, or the lack of adherence to best practices. Studies have shown that code smells can significantly contribute to this technical debtTufano et al. [2015]. One of the challenges is that code smells often emerge gradually over the software's lifecycle, as developers introduce suboptimal design choices during maintenanceTufano et al. [2015]. Detecting and addressing these issues early can greatly reduce

the time and effort spent on future maintenance Santos et al. [2018].

Although many tools have been developed to detect code smells, including static analysis tools and machine learning-based approaches, no solution has completely solved the problemSahin et al. [2014]Tufano et al. [2015]. Most tools rely on predefined rules or metrics that require manual calibration, which can lead to false positives or the inability to detect complex smell patternsSahin et al. [2014]. Recent advances, such as bi-level optimization techniques and machine learning-based solutions, have shown promise in improving detection accuracy and generalizabilitySahin et al. [2014]. However, further improvements are still necessary to enhance precision and reduce false positives, making this an ongoing area of research.Santos et al. [2018]

Business

The business need for a code quality tool like CodePure is evident in critical fields such as software development, where maintaining clean, efficient, and scalable code is essential for long-term success. CodePure can be used to automate code smell detection, which allows development teams to improve code quality and reduce technical debt without manual effort.

In addition, CodePure is highly valuable in maintaining high software standards in industries where compliance with coding best practices is essential, ensuring that businesses deliver reliable software. Moreover, it assists in managing large-scale software systems by identifying issues early, allowing businesses to sustain growth and add new features seamlessly.

1.1.1 Problem Statement

Software systems often suffer from poor design decisions that accumulate over time, leading to the presence of code smells. These issues not only affect the maintainability and scalability of the code but also introduce technical debt that becomes costly to address. Despite the availability of several code smell detection tools, two key challenges remain

unresolved.**The first challenge is the accuracy of code smell detection**. Current tools tend to rely on rigid predefined rules and metrics that often result in high false positive and false negative rates. This diminishes their effectiveness, especially when applied to different codebases with varying styles and structures. There is a need for a solution that can accurately detect a wide range of code smells while adapting to the unique characteristics of each project.**The second challenge is the lack of real-time feedback during the development process**. Most existing tools provide feedback after the code has been committed or in later stages of the development lifecycle, making it harder to address issues promptly. This project aims to develop an IDE extension that integrates seamlessly into the developer's workflow, offering real-time code smell detection as the developer writes code. By providing immediate feedback, this extension can help reduce technical debt early in the development process, improving overall code quality.

1.2 Aims and Objectives

The primary objective of the *CodePure* extension is to assist developers in writing clean, maintainable, and high-quality code. This is achieved by providing real-time detection and classification of code smells, offering actionable refactoring suggestions, and empowering developers to manage technical debt effectively. The extension aims to reduce false positives in code smell detection through advanced analysis and customizable thresholds. Additionally, *CodePure* aspires to improve workflow efficiency by integrating seamlessly with Integrated Development Environments (IDEs). The ultimate goal is to enhance software maintainability, scalability, and adherence to coding best practices.

1.3 Scope

The scope of the IDE extension (CodePure) for code smell detection includes the following:

- **Real-time Detection**: The extension will provide real-time feedback, identifying code smells within seconds of typing or modifying code.

- **Supported Languages**: The extension will support code smell detection for multiple programming languages.
- **AI Model Training**: The AI model will be extensively trained on a comprehensive dataset to ensure high detection accuracy.
- **User Interface:** Provides a visual representation of detected code smells, offering actionable insights for improvement in a dashboard.
- **Automated Refactoring Suggestions:** Offers quick fixes and detailed guidance for resolving detected issues.
- **Seamless IDE Integration:** Works directly within Visual Studio Code, integrating into the developer's workflow for maximum efficiency.
- **Performance Efficiency**: The extension will be optimized to run efficiently, without significantly affecting the performance of the IDE.

1.4 System Overview

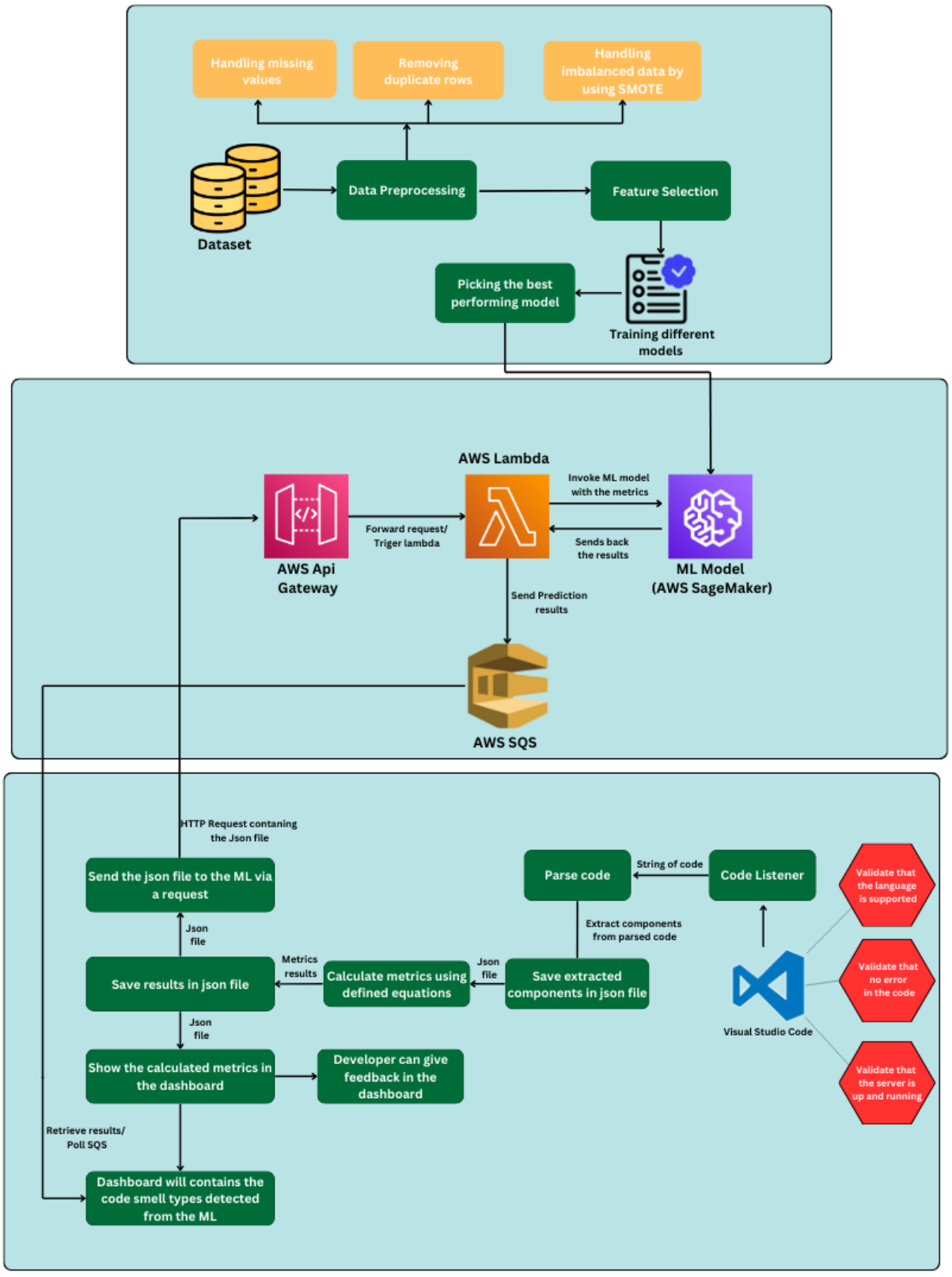


Figure 1.1: Detailed System Overview

The **CodePure** system integrates machine learning and cloud computing to analyze code quality, detect code smells, and provide feedback to developers. The architecture is divided into three main sections: **Data Preprocessing & Model Training, AWS Cloud Infrastructure, and Code Analysis & Dashboard**.

1.4.1 Data Preprocessing & Model Training

This section focuses on preparing data and training machine learning models:

- **Dataset Preparation:**
  - Handling missing values.
  - Removing duplicate rows.
  - Addressing data imbalance.
- **Feature Selection & Model Training:**
  - Selecting features from the dataset.
  - Training multiple machine learning models.
  - Selecting the best-performing model for deployment.

1.4.2 AWS Cloud Infrastructure

Once the best model is selected, it is deployed using AWS services:

- **AWS API Gateway:** Accepts requests for code analysis.
- **AWS Lambda:** Processes requests and interacts with the ML model.
- **AWS SageMaker:** Hosts the trained machine learning model for prediction.
- **AWS SQS:** Manages the communication between different AWS components.

1.4.3 Code Analysis & Dashboard

This section details how the **CodePure** system processes developer code and presents insights:

- **Code Collection & Validation:**
  - Collects code for analysis.
  - Validates the programming language.
  - Ensures there are no syntax errors.
- **Code Parsing & Component Extraction:**
  - Parses the code.
  - Extracts key components for analysis.
- **Metric Calculation & ML Processing:**
  - Computes software quality metrics using predefined equations.
  - Sends data to the ML model via JSON request.
- **Results & Developer Feedback:**
  - Displays calculated metrics on a dashboard.
  - Developers can provide feedback on detected issues.
  - The dashboard also presents detected code smell types.

1.4.4 Business Context

The business need for a code quality tool like CodePure is evident in critical fields such as software development, where maintaining clean, efficient, and scalable code is essential for long-term success. CodePure can be used to automate code smell detection, which allows development teams to improve code quality and reduce technical debt without manual effort.

In addition, CodePure is highly valuable in maintaining high software standards in industries where compliance with coding best practices is essential, ensuring that businesses deliver reliable software. Moreover, it assists in managing large-scale software systems by identifying issues early, allowing businesses to sustain growth and add new features seamlessly.

1.4.5 Users Characteristics

The users of the **CodePure** extension are software developers, including individual developers and teams, who work on maintaining or improving the quality of their code. The extension is designed to cater to the following user characteristics:

- **Experience Level:** Users range from beginner developers to experienced professionals. The system is designed to be intuitive and easy to use for novices, while providing advanced features for experienced developers to assist them in improving their code quality.
- **Programming Languages:**The extension supports multiple programming languages to detect code smells based on predefined metrics.
- **Project Types:** The extension is designed to work on small-scale and medium scale projects. It can be utilized for individual projects or as part of a collaborative team environment.
- **Expectations:** Users expect the extension to provide timely and accurate feedback on code quality and offer suggestions for improvement. They also expect the system to be highly reliable and to integrate smoothly into their existing development workflow.
- **Technical Proficiency:** Users are expected to be familiar with basic IDE usage, code analysis tools, and software development concepts, although detailed knowledge of code smell metrics and machine learning models is not required.

By catering to developers of varying experience levels and offering a user-friendly interface, **CodePure** aims to enhance the development process, providing users with valuable insights that help improve code maintainability and overall software quality.

1.5 Project Management and Deliverables

1.5.1 Time Plan

Task	Start date	End date	Duration	Assign
Ideas and Supervisor	8/09/2024	12/09/2024	5 Days	All
Information collection and research	13/09/2024	24/9/2024	12 days	All
Survey and proposal preparation	25/09/2024	5/10/2024	11 Days	All
SRS preparation	10/10/2024	1/11/2024	23 Days	All
SRS presentation	1/11/2024	4/11/2024	4 Days	All
SDD preparation	6/11/2024	28/11/2024	23 Days	All
SDD presentation	28/11/2024	2/12/2024	4 Days	All
Data preprocessing and AI Model training	4/12/2024	1/01/2025	27 Days	All
Website development	2/1/2025	10/01/2025	8 Days	All
Extension development	2/1/2025	5/04/2025	93 Days	All
Prototype	2/1/2025	20/03/2025	77 Days	All
Testing and validating	25/03/2025	5/05/2025	41 Days	All
Thesis	25/06/2025	28/06/2025	3 Days	All

Table 1.1: Time and plan table

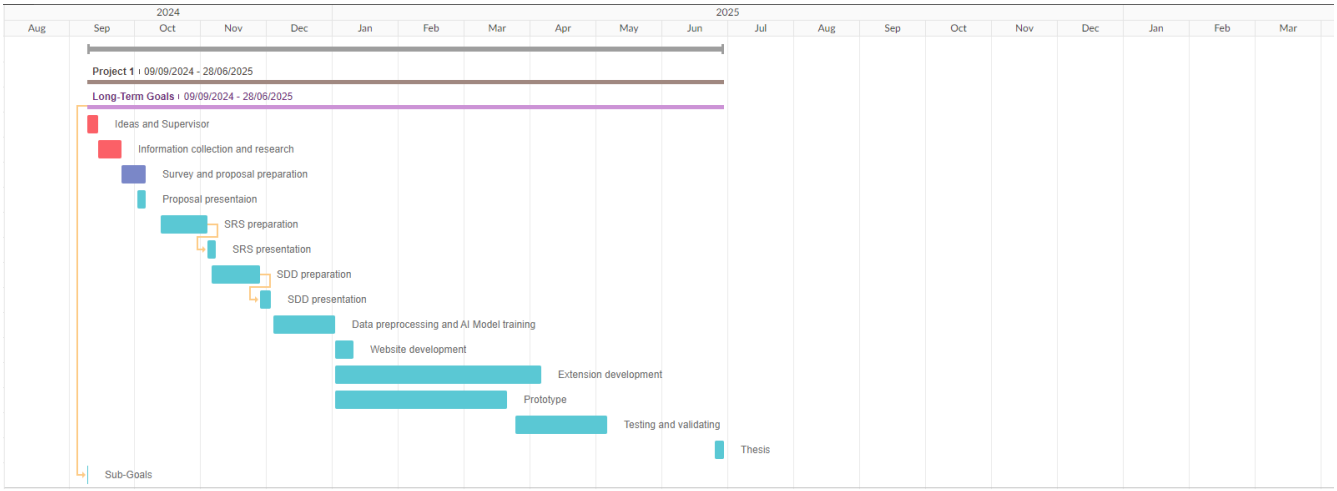


Figure 1.2: Gantt chart

1.5.2 Deliverables

The project will develop an IDE extension that helps developers identify code smells as they write their code. This extension will provide real-time feedback, highlighting potential issues and offering suggestions for improvement.

- Proposal Document.
- SRS Document.
- SDD Document.
- Thesis.
- Research paper
- CodePure extension.
- Website.

1.6 Thesis Overview

This thesis is organized into the following chapters:

- **Chapter 1: Introduction** — Provides an overview of the project, including motivation, problem statement, aims and objectives, scope, system overview, and project management.
- **Chapter 2: Background** — Reviews existing research and technologies related to code smell detection.
- **Chapter 3: System Requirements and Analysis** — Describes the functional and non-functional requirements, and the system analysis process.
- **Chapter 4: System Design and Implementation** — Details the system architecture, design choices, machine learning model training, and integration with Visual Studio Code.
- **Chapter 5: Evaluation and Testing** — Presents the methods used to test and validate the system, including performance evaluation and analysis.
- **Chapter 6: Conclusion and Future Work** — Summarizes the research outcomes, discusses limitations, and suggests directions for future enhancements.

Additional sections include the abstract, acknowledgements, bibliography, and appendices.

Chapter 2

Background and Related Work

2.1 Introduction

The **CodePure** extension provides a novel solution to the ongoing challenge of maintaining high-quality code by offering real-time code smell detection directly within the Visual Studio Code (VSCode) IDE. Unlike existing tools that typically require static analysis or external integrations, **CodePure** delivers immediate feedback while developers are coding, allowing them to address potential issues as they arise. Additionally, by incorporating a machine learning model for categorizing and suggesting improvements for code smells, the extension offers more precise and context-aware insights compared to traditional rule-based systems. With support of multiple languages, real-time analysis, historical tracking of code smells, and seamless IDE integration, **CodePure** stands apart from other tools by providing a comprehensive and user-friendly solution that enhances the development process and promotes cleaner, more maintainable code.

2.2 Background

Today, software development is essential to practically every industry, serving as the basis for applications that drive anything from intricate corporate processes to web services. It gets harder to guarantee software systems' quality and maintainability as they expand and change over time. How to control code complexity over time to preserve good per-

formance and scalability is a big worry for developers.

One of the main areas of attention in the larger field of software engineering is software quality. Reducing errors, enhancing readability, and making sure software systems can adjust to changes over time all depend on maintaining high-quality code. In the course of this endeavour, developers and academics have examined a number of variables that affect code quality, one of which is technical debt—the price paid for using coding techniques that aren't ideal and increase the difficulty of future development and maintenance.

Code smells are a particular kind of technical debt. They are patterns in the code that, although not necessarily defects, suggest possible issues that can cause problems down the road. These "smells" frequently indicate that the code is ineffective, excessively complicated, or badly organised. Long methods, big classes, and redundant code are common examples. Early detection and correction of code smells ensures that the code is clear, manageable, and simple to refactor.

The goal of our project, CodePure, is to automatically identify these code smells in Visual Studio Code extension. By giving developers immediate feedback on the quality of their code, the add on hopes to facilitate the identification and correction of problems as developers create code. With CodePure, developers can customize criteria, code metrics, and datasets to meet their unique requirements for the detection process. By doing this, we hope to support the continuous work in software quality assurance by providing a flexible, multi-language solution for managing codebases that are cleaner and more productive.

2.3 Related Systems

2.3.1 Academic

Di Nucci et al. [2018] addressed the main problem of subjectivity and inconsistency in the results produced by traditional code smell detection tools in their paper titled "Detecting Code Smells using Machine Learning Techniques: Are We There Yet?". The researchers aim to explore whether machine learning (ML) techniques can offer a more objective and reliable approach to detecting code smells. To tackle this problem, the au-

thors conducted a replication study by using a different dataset configuration to evaluate the capabilities of ML techniques under more realistic conditions. They contributed by expanding the experimental setup to include datasets containing instances of multiple types of smells, rather than focusing on single-smell datasets as done in prior studies. The dataset used comprised 74 Java software systems, initially developed for the Qualitas Corpus project. The primary result of the study was that, under a more realistic dataset setting, the performance of ML techniques dropped significantly—up to 90% in F-measure compared to the previous study—suggesting that the problem of using ML for accurate code smell detection remains unsolved. One critical observation is that the study successfully highlights the shortcomings of ML in real-world scenarios, but it would have been more insightful if the authors proposed potential improvements or hybrid approaches to overcome these limitations.

Fontana et al. [2012] The paper titled "Automatic detection of bad smells in code: An experimental assessment" addresses the inconsistency in code smell detection across different automatic tools, a problem that makes it challenging for developers to rely on these tools for software maintenance. The researchers contributed by experimentally evaluating four widely-used detection tools on various versions of the GanttProject software, assessing how each tool performs in identifying code smells. Using this open-source dataset, they found significant variations between the tools, with minimal agreement on detected smells, thus showing that these tools provide inconsistent results. While the paper effectively highlights the limitations in current detection technologies, it falls short in proposing specific solutions to improve the precision and agreement between these tools. Additionally, the dataset used is limited in scope, making the results less generalizable.

Paiva et al. [2017] The paper "On the Evaluation of Code Smells and Detection Tools" tackles the problem of inconsistencies and lack of agreement between different code smell detection tools, which complicates the trustworthiness of these tools for software developers. The authors conducted an empirical assessment of four detection tools (inFusion, JDeodorant, PMD, and JSpIRIT) using two open-source projects, MobileMedia and

Health Watcher, to evaluate their precision, recall, and agreement in detecting God Class, God Method, and Feature Envy smells. The study found wide variations in performance, with recall ranging from 0% to 100%, suggesting that the tools are unreliable in their current form. Although the paper does well in pointing out discrepancies between tools, it offers little in terms of actionable recommendations for improving their consistency.

Code Smell	inFusion		JDeodorant		JSpIRIT		PMD		Reference List	
	MM	HW	MM	HW	MM	HW	MM	HW	MM	HW
God Class	3	0	85	98	9	20	8	33	47	12
God Method	17	0	100	599	27	30	16	13	67	60
Feature Envy	8	48	69	90	74	111	–	–	19	0
Total	28	48	254	787	110	161	24	46	133	72

Figure 2.1: Total number of code smells detected by each tool

Code Smell	inFusion		JDeodorant		JSpIRIT		PMD	
	R	P	R	P	R	P	R	P
God Class	9%	33%	58%	28%	17%	67%	17%	78%
God Method	26%	100%	50%	35%	36%	93%	26%	100%
Feature Envy	0%	0%	48%	13%	0%	0%	n/a	n/a

Code Smell	inFusion		JDeodorant		JSpIRIT		PMD	
	R	P	R	P	R	P	R	P
God Class	0%	undef.	70%	8%	10%	10%	100%	36%
God Method	0%	undef.	82%	8%	17%	33%	17%	85%
Feature Envy	undef.	0%	undef.	0%	undef.	0%	n/a	n/a

Figure 2.2: Average recall and precision for MobileMedia and Health Watcher

Azeem et al. [2019] The paper titled "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis" addresses the limitations of traditional heuristic-based code smell detection methods, which are often subjective and inconsistent, by investigating the use of machine learning (ML) techniques to enhance detection accuracy. The researchers conducted a systematic literature review of studies

published between 2000 and 2017, focusing on the effectiveness of various ML models such as Decision Trees and Support Vector Machines, using datasets derived from these studies. Their meta-analysis revealed that JRip and Random Forest were the most effective classifiers, but also identified key gaps such as that a wrong selection of the machine learning algorithm to use impacts the performance of code smell prediction models by up to 40% in terms of F-Measure and a lack of standardized datasets. While the paper provides a comprehensive review, it lacks concrete suggestions for overcoming these challenges and does not incorporate more recent advancements, limiting the relevance of its conclusions.

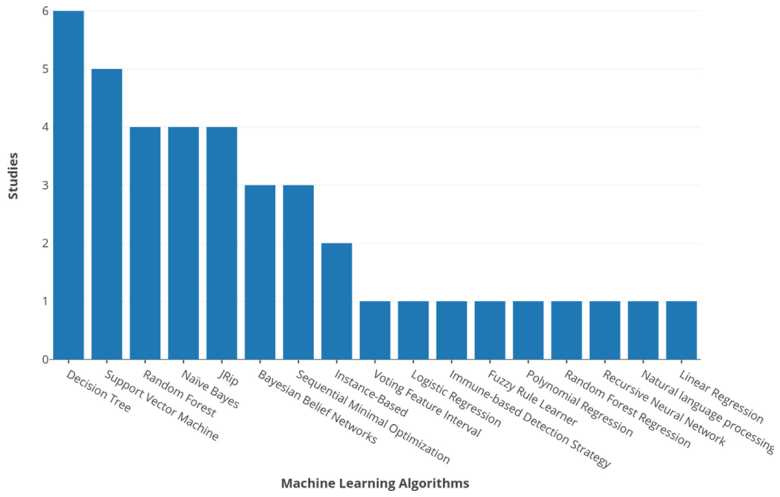


Figure 2.3: Frequency of each machine learning algorithm across the primary studies.

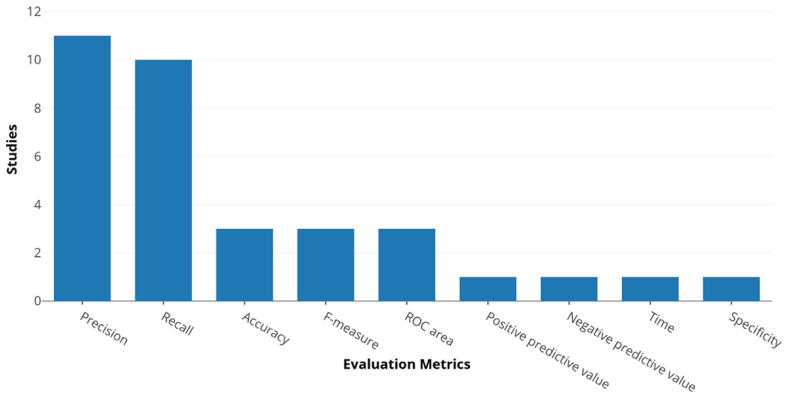


Figure 2.4: Evaluation metrics used across the primary studies.

Madeyski and Lewowski [2023] The paper titled "Detecting code smells using industry-relevant data" addresses the problem of detecting code smells, which are patterns in

software code associated with increased defects and higher maintenance effort, typically identified using predefined, arbitrary rules. These rules often lack industrial relevance, leading to unreliable conclusions. To solve this, the researchers evaluated the performance of eight machine learning algorithms on a dataset (MLCQ) built from industry-relevant projects reviewed by experienced developers. The dataset contained over 14,000 reviews of code snippets labeled with four types of code smells: Blob, Data Class, Feature Envy, and Long Method. The study found that the Random Forest and Flexible Discriminant Analysis algorithms performed the best, with performance varying depending on the type of smell. The highest median MCC value was 0.81 for detecting Long Method, while the lowest was 0.31 for Feature Envy. Although the study provides valuable insights into using machine learning for code smell detection, it is limited by the lack of diversity in predictors and the reliance on proprietary metrics, which could hinder reproducibility and generalization in future research. Moreover, the performance differences between algorithms were minimal, indicating the need for new predictive features to improve detection.

Table A.4
Ranking of machine algorithms for each of code smell data sets (based on median MCC) - higher is better.

Smell	DS	CTree	FDA	KNN	kSVM	libSVM	MDA	Naïve Bayes	Random Forest
Blob	DS1	5	6	1	3	4	0	2	7
Blob	DS2	3	7	4	0	2	6	1	5
Data Class	DS1	5	6	2	3	4	1	0	7
Data Class	DS2	4	6	1	2	5	3	0	7
Long Method	DS1	4	5	0	3	1	2	6	7
Long Method	DS2	4	6	5	3	1	2	0	7
Feature Envy	DS1	6	7	2	0	3	4	1	5
Total	–	31	43	15	14	20	18	10	45

Table A.5
Ranking of machine algorithms for each of code smell data sets (based on median F-score) - higher is better.

Smell	DS	CTree	FDA	KNN	kSVM	libSVM	MDA	Naïve Bayes	Random Forest
Blob	DS1	5	6	1	4	3	0	2	7
Blob	DS2	3	5	4	0	2	7	1	6
Data Class	DS1	5	6	2	4	3	1	0	7
Data Class	DS2	5	6	1	3	4	2	0	7
Long Method	DS1	3	5	1	2	0	4	6	7
Long Method	DS2	4	6	5	3	1	2	0	7
Feature Envy	DS1	4	7	2	1	3	5	0	6
Total	–	29	41	16	17	16	21	9	47

justification=centering

Figure 2.5: Ranking of machine algorithms for each of code smell datasets based on median MCC and median F-score

Fontana et al. [2011]The paper titled "An Experience Report on Using Code Smells Detection Tools" explores the effectiveness of various code smell detection tools by comparing their results on multiple versions of the GanttProject software. The main problem addressed is the inconsistency and ambiguity in detecting code smells using different tools, which complicates code quality evaluation and maintenance. The researchers conducted an experimental evaluation using five popular tools: JDeodorant, Stench Blossom, InFusion, iPlasma, and PMD, and compared their effectiveness on detecting code smells like God Class, Long Method, and Feature Envy. The dataset consisted of multiple versions of GanttProject, with detailed comparisons based on metrics like number of packages, classes, and methods. The main result showed significant differences in the detection capabilities of these tools, with some tools like JDeodorant and Stench Blossom performing inconsistently across different releases of the same software. The paper emphasizes that there is no single "best" tool for all code smells, as results vary based on the detection rules and metric thresholds.

Hadj-Kacem and Bouassida [2018] The paper titled "A Hybrid Approach to Detect Code Smells using Deep Learning" addresses the problem of detecting code smells, which are structural weaknesses in software code that indicate deeper design issues, leading to high maintenance costs and reduced software quality. To tackle this, the authors propose a hybrid method combining an unsupervised deep auto-encoder for dimensionality reduction with a supervised artificial neural network (ANN) for classification. The approach is tested on four types of code smells—God Class, Data Class, Feature Envy, and Long Method—using datasets derived from 74 open-source projects. The results demonstrate high accuracy, with precision and recall values exceeding 96%, highlighting the importance of feature reduction for enhancing detection performance.

Guggulothu and Moiz [2020]The paper titled "Code Smell Detection Using Multi-Label Classification Approach" addresses the challenge of detecting multiple overlapping code smells in software, which are critical indicators of design flaws impacting maintainability. Traditional approaches focus on single-label detection and fail to consider the cor-

relation among smells. To tackle this, the researchers propose a multi-label classification (MLC) approach, enabling simultaneous detection of multiple code smells while accounting for their correlation. They created a multi-label dataset by combining method-level code smells, "Long Method" and "Feature Envy," using data from 74 open-source Java projects, balancing it to address disparity instances that previously reduced classifier performance. Experimental results showed that incorporating correlation through MLC techniques, particularly Classifier Chains and Label Combination, improved detection accuracy to over 96%, significantly outperforming single-label approaches. The findings highlight the potential of MLC for realistic, accurate, and practical code smell detection, offering a valuable tool for prioritizing refactoring tasks in software development.

LC (10-fold cross-validation run for 10 iterations)

Single label classifier	Example-based metrics		
	Accuracy (Jaccard index)	Hamming loss	Exact match
J48 Pruned	96.7%	0.02	96%
Random Forest	96.7%	0.024	95.4%
B-J48 pruned	*97.5%*	*0.016*	*96.9%*
B-J48 UnPruned	97.2%	0.018	96.4%
B-Random Forest	96.7%	0.024	95.4%

CC (10-fold cross-validation run for 10 iterations)

Single label classifier	Example-based metrics		
	Accuracy (Jaccard index)	Hamming loss	Exact mat
J48 Pruned	96.7%	0.024	95.3%
Random Forest	96.8%	0.022	95.6%
B-J48 pruned	96.5%	0.023	95.3%
B-J48 UnPruned	97%	0.022	95.5%
B-Random Forest	*97%*	*0.021*	*95.8%*

BR (10-fold cross-validation run for 10 iterations)

Single label classifier	Example-based metrics		
	Accuracy (Jaccard index)	Hamming loss	Exact match
J48 Pruned	*96.3%*	*0.028*	*94.4%*
Random Forest	93.7%	0.044	91.9%
B-J48 pruned	94.6%	0.037	92.6%
B-J48 UnPruned	95%	0.035	93.1%
B-Random Forest	93.6%	0.044	91.8%

Figure 2.6: Results of LC & CC & BR using top 5 label classifiers

2.3.2 Business Applications

Hammad and Labadi [2016] The paper titled "Automatic Detection of Bad Smells from Code Changes" addresses the problem of maintaining code quality by detecting "bad

smells," which are indicators of poor design choices that affect code maintainability and readability. The authors propose a lightweight, real-time approach to automatically detect these smells as code changes are implemented using an Eclipse plug-in tool called JFly. The tool monitors code changes, calculates relevant software metrics, and applies predefined detection rules to identify nine different bad smells, such as Long Parameter List, Brain Method, and Lazy Class. The dataset used includes multiple Java projects, such as Payments Auditing and Smart Payment Router for performance testing, and three industrial projects (Payment Gateway, Settlement, and Fees) for correctness evaluation. The results show that JFly performs efficiently, with an average detection time of less than three seconds, and achieves high precision and recall values in detecting bad smells, especially for Long Parameter List and Brain Method. However, a key limitation is that the approach does not account for larger codebases and might struggle with complex dependencies between classes, which led to some false positives and false negatives during real-world testing.

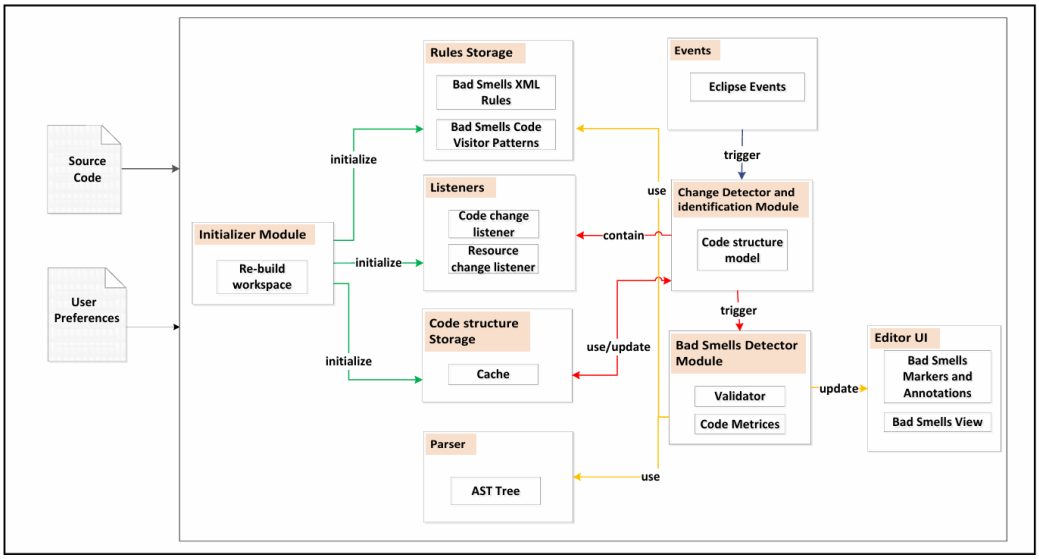


Figure 2.7: JFly plug-in Architecture

PRECISION AND RECALL FOR PAYMENT GATEWAY

Bad Smell	E existing Bad Smells	Detected Bad Smells	True Pos.	Precision	Recall
LPL	9	9	9	100%	100%
BM	22	28	22	78.6%	100%
TF	21	29	17	58.6%	81%
DC	8	10	8	80%	100%
MM	6	7	6	85.7%	100%
MC	5	6	5	83.3%	100%
SG	43	51	35	68.6%	81.4%
II	27	32	20	62.5%	74%
LC	19	21	16	76.2%	84.2%
No. of Changed Classes = 73; No. of commits = 74					

PRECISION AND RECALL FOR SETTLEMENT

Bad Smell	Existing Bad Smells	Detected Bad Smells	True Pos.	Precision	Recall
LPL	8	8	8	100%	100%
BM	13	17	12	70.6%	92.3%
TF	17	24	13	54.1%	76.5%
DC	4	4	4	100%	100%
MM	4	6	3	50%	75%
MC	22	25	17	68%	77.3%
SG	26	35	22	62.9%	84.6%
II	15	23	12	52.2	80%
LC	10	13	10	76.9	100%
No. of changed classes =43; No. of commits = 44					

PRECISION AND RECALL FOR FEES

Bad Smell	Existing Bad Smells	Detected Bad Smells	True Pos.	Precision	Recall
LPL	-	-	-	-	
BM	10	10	10	100%	100%
TF	14	14	14	100%	100%
DC	1	2	1	50%	100%
MM	10	13	10	71.4%	100%
MC	5	8	5	62.5%	100%
SG	18	32	16	50%	88.9%
II	5	9	5	55.5%	100%
LC	11	15	11	73.3%	100%
No. of changed classes = 32; No. of commits = 38					

Figure 2.8: Precision and Recall Results

Van Emden and Moonen [2002] The paper titled "Java Quality Assurance by Detecting Code Smells" addresses the problem of inefficient and labor-intensive traditional software inspection methods that fail to scale for large systems. The authors propose automating the detection and visualization of code smells to make software quality assurance more practical and efficient. They contributed by developing a tool named jCOSMO, a prototype code smell detector and visualizer for Java code, which automates the inspection process by identifying design and coding issues such as "large classes" or "typecasts" and visualizing these smells to assist developers in understanding and improving code quality. The dataset used for evaluation was the source code of a Java-based system named CHARTOON, consisting of 147 classes and 46,000 lines of code. The main results show that jCOSMO successfully identified code smells and provided useful visualizations that enabled maintainers to pinpoint problematic areas and suggest refactoring. However, a limitation of the paper is that it only presents a qualitative evaluation through a single

case study without quantitative performance metrics, making it difficult to generalize the tool's effectiveness across various software projects. Additionally, while the tool provides valuable visualization capabilities, the authors could have extended the work to automate more advanced refactoring suggestions based on detected smells.

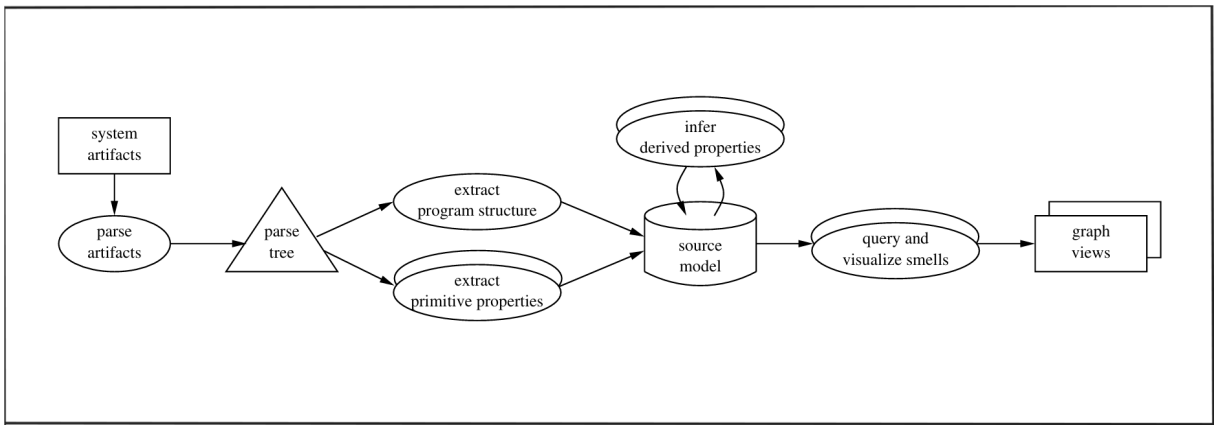


Figure 2.9: Architecture of code smell browser

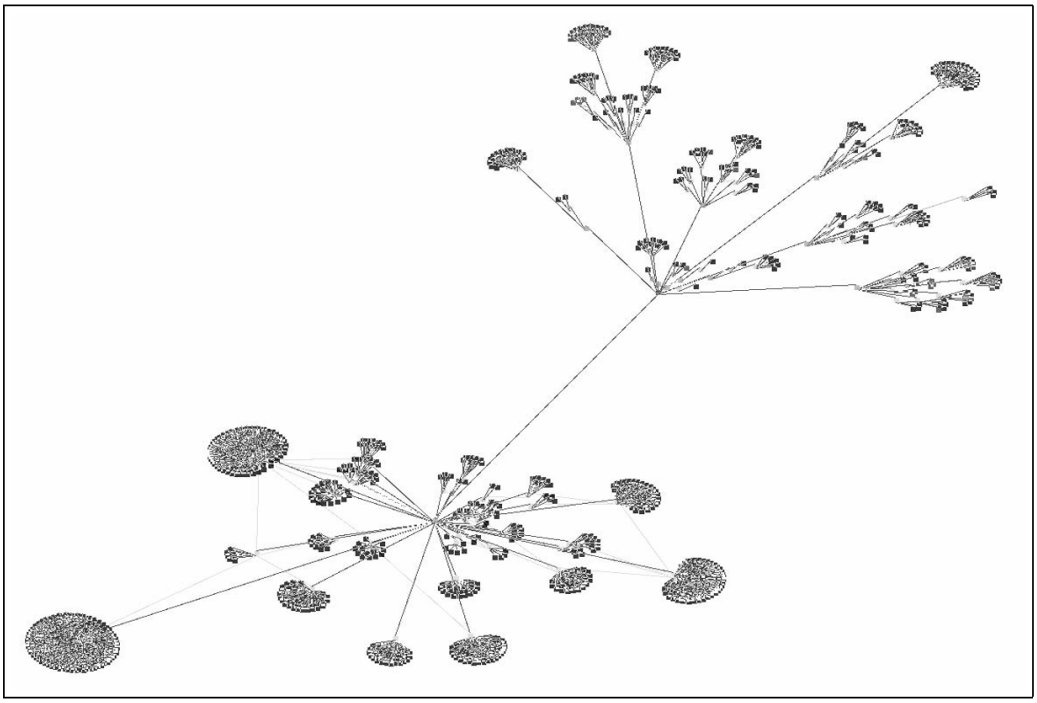


Figure 2.10: Complete smell detection graph for a small test system (12 classes, 1450 LOC)

2.4 Summary

Several research papers and related systems were reviewed to establish the context of existing solutions in the field of code quality analysis. These studies highlight the challenges faced by current tools and demonstrate how **CodePure** distinguishes itself by offering more comprehensive, real-time feedback and seamless IDE integration. The insights gained from these papers provided valuable input in shaping the design and features of **CodePure**, ensuring that it addresses gaps left by existing systems while pushing the boundaries of traditional code analysis tools.

Chapter 3

System Requirements Specification

3.1 Introduction

This chapter outlines the software requirements for CodePure, a Visual Studio Code extension that helps developers identify and examine code smells in their own code. The chapter defines the Functional Requirements, detailing the core capabilities of the extension; Interface Requirements, specifying interactions between users and the system; Design Constraints, addressing limitations that influence development; and Non-functional Requirements, ensuring performance, usability, and reliability. This chapter attempts to support successful cooperation and the delivery of a high-quality software solution by outlining requirements in a clear and succinct manner.

3.2 Functional Requirements

3.2.1 System Functions

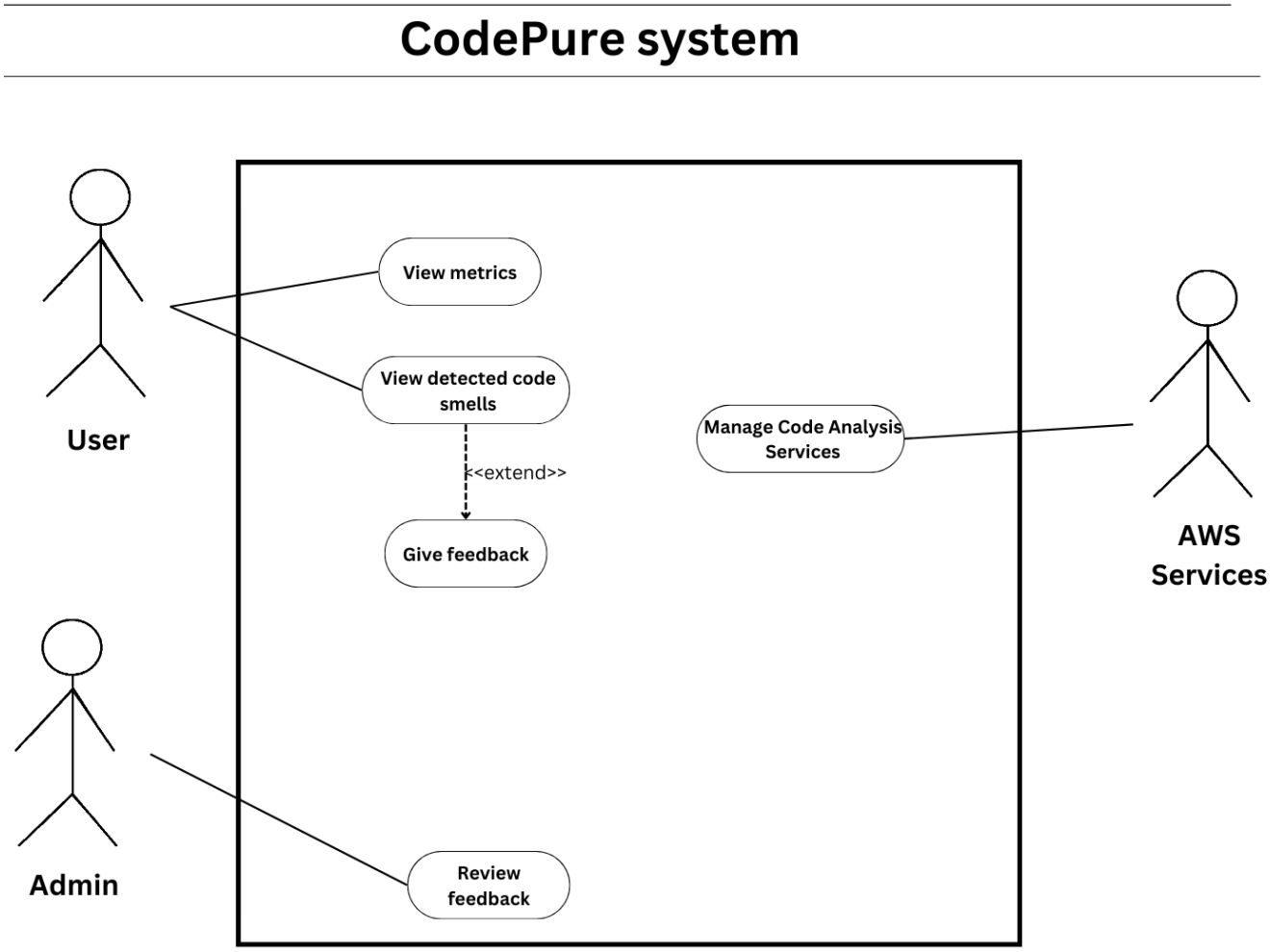


Figure 3.1: Use Case Diagram

- **Id:01** The user shall be able to view detected code smells in the system.
- **Id:02** The user shall be able to view various code metrics on the dashboard.
- **Id:03** The user shall be able to access the dashboard to monitor their code analysis progress.
- **Id:04** The user shall be able to provide feedback on the smells or metrics detected in the code.
- **Id:05** The admin shall be able to review user-provided feedback for system improvements.
- **Id:06** The extension shall parse the code provided by the user to extract relevant details.
- **Id:07** The extension shall calculate specific code metrics based on the parsed data.
- **Id:8** AWS shall receive the calculated code metrics from the extension.
- **Id:9** AWS shall manage the deployment of the machine learning model.
- **Id:10** AWS shall process and retrieve detection results from the ML model.
- **Id:11** AWS shall send the detection results back to the extension for further use.
- **Id:12** The extension shall update the dashboard with the latest detection results and analysis data.

3.2.2 Detailed Functional Specification

Table 3.1: View Detected Code Smells

Name	View Detected Code Smells
Code	FR01
Priority	High
Critical	Essential for displaying detected issues to the user.
Description	This requirement allows the user to view the code smells that have been detected in their code. It provides an interface to present the identified issues in an understandable format.
Input	Detected code smells data (e.g., smell type, location, severity).
Output	List of detected code smells with details.
Pre-condition	The code must have been analyzed and code smells detected.
Post-condition	The user can view and understand the code smells detected in their code.
Dependency	Requires the code parsing and analysis to be performed.
Risk	If the analysis fails, no code smells will be detected and displayed.

Table 3.2: View Code Metrics on Dashboard

Name	View Code Metrics on Dashboard
Code	FR02
Priority	Medium
Critical	Important to track and monitor code quality metrics.
Description	This requirement enables the user to view various code metrics such as Lines of Code, Cyclomatic Complexity, and Method Count on the dashboard.
Input	Metrics data from code analysis (e.g., LOC, CC, Method Count).
Output	Display of metrics on the dashboard.
Pre-condition	Metrics must be calculated and ready for display.
Post-condition	User sees the up-to-date code metrics displayed on the dashboard.
Dependency	Dependent on the completion of code analysis and metric calculation.
Risk	If metrics calculation fails, metrics will not be shown.

Table 3.3: Access Dashboard for Code Monitoring

Name	Access Dashboard for Code Monitoring
Code	FR03
Priority	High
Critical	Vital for the user to monitor ongoing code quality analysis.
Description	This requirement enables users to access the dashboard, where they can monitor the progress of their code analysis in real-time.
Input	User request to view dashboard.
Output	Display of the dashboard with up-to-date analysis data.
Pre-condition	User must be authenticated and have data available for display.
Post-condition	User sees the dashboard with the latest code analysis status.
Dependency	Relies on system status and data availability from analysis processes.
Risk	System downtime or lack of data could prevent dashboard access.

Table 3.4: Provide Feedback on Detected Code Smells

Name	Provide Feedback on Detected Code Smells
Code	FR04
Priority	Medium
Critical	Important for users to give feedback on detected issues for further improvements.
Description	This requirement allows the user to provide feedback on the detected code smells, which can help improve the detection system or suggest false positives/negatives.
Input	User feedback on detected code smells.
Output	Feedback submitted to the system for review.
Pre-condition	User must be logged in and have detected code smells to provide feedback on.
Post-condition	Feedback is stored for review and potential system improvement.
Dependency	Requires the feedback system and database to store responses.
Risk	If feedback is not stored correctly, it may not be useful for future improvements.

Table 3.5: Admin Review of User Feedback

Name	Admin Review of User Feedback
Code	FR05
Priority	High
Critical	Critical for ensuring continuous improvement of the detection system.
Description	This requirement allows the system administrator to review the feedback provided by users and make necessary changes or improvements based on the feedback.
Input	User feedback data.
Output	Processed feedback and insights for system improvements.
Pre-condition	Feedback from users must be collected and stored.
Post-condition	Admin has reviewed feedback and initiated actions for improvement.
Dependency	Relies on the feedback collection system and the admin interface.
Risk	If feedback is not reviewed, the system may fail to improve based on user input.

Table 3.6: Parse Code for Relevant Details

Name	Parse Code for Relevant Details
Code	FR06
Priority	High
Critical	Essential for the extension to extract necessary details from the code for further analysis.
Description	This requirement involves parsing the user's code to extract relevant details such as function/method definitions, class structures, and other code elements necessary for detecting code smells.
Input	Source code provided by the user.
Output	Parsed code details such as method counts, class structures, etc.
Pre-condition	The user must provide the code for analysis.
Post-condition	The code is parsed and ready for further processing.
Dependency	Depends on the parsing library and code analysis framework.
Risk	If parsing fails, the system cannot proceed with further analysis.

Table 3.7: Calculate Code Metrics

Name	Calculate Code Metrics
Code	FR07
Priority	High
Critical	Important for evaluating code quality.
Description	This requirement involves calculating specific code metrics, such as Lines of Code (LOC), Cyclomatic Complexity (CC), and Method Count, based on the parsed data. These metrics are crucial for identifying code smells.
Input	Parsed code details.
Output	Calculated metrics such as LOC, CC, and Method Count.
Pre-condition	Code must be parsed, and metrics must be definable.
Post-condition	Calculated metrics are ready for submission to the machine learning model.
Dependency	Dependent on code parsing and the metric calculation methods.
Risk	Incorrect parsing or metric calculation could lead to erroneous results.

Table 3.8: Send Metrics to ML Model

Name	Send Metrics to ML Model
Code	FR08
Priority	High
Critical	Essential for the model to receive data for code smell detection.
Description	This requirement involves sending the calculated metrics to the integrated machine learning model for further analysis to detect code smells.
Input	Calculated metrics data (e.g., LOC, CC, Method Count).
Output	ML model receives and processes metrics data.
Pre-condition	Metrics must be calculated and ready for submission.
Post-condition	Metrics are processed by the ML model.
Dependency	Relies on the availability of the ML model and data processing system.
Risk	If the ML model is unavailable or unable to process metrics, detection will fail.

Table 3.9: Receive Detection Results from ML Model

Name	Receive Detection Results from ML Model
Code	FR09
Priority	High
Critical	Essential for the system to receive predictions for further processing.
Description	This requirement involves receiving the results of the code smell detection from the machine learning model after processing the metrics.
Input	Detection results from the ML model.
Output	Results of the detection process, including code smell type and severity.
Pre-condition	ML model must return results after processing the metrics.
Post-condition	Detection results are ready for user presentation.
Dependency	Dependent on the ML model output.
Risk	Incorrect or delayed results from the model can hinder user experience.

Table 3.10: Return Prediction Results to User

Name	Return Prediction Results to User
Code	FR10
Priority	High
Critical	Crucial for presenting results to users.
Description	This requirement ensures that the results from the machine learning model are returned to the user in an understandable format, providing clear information on the detected code smells and suggestions for remediation in the dashboard.
Input	Detection results from the ML model.
Output	User-friendly presentation of code smell predictions in the dashboard.
Pre-condition	Results must be received from the ML model.
Post-condition	User views the detection results in a comprehensible format in the dashboard.
Dependency	Relies on proper formatting of results from the ML model.
Risk	If results are not presented clearly, user understanding may be affected.

Table 3.11: Update Dashboard with Latest Detection Results

Name	Update Dashboard with Latest Detection Results
Code	FR11
Priority	High
Critical	Important for displaying up-to-date results to the user.
Description	This requirement ensures that the dashboard is updated with the latest code smell detection results and metrics, keeping the user informed of their code's quality.
Input	Latest detection results and analysis data.
Output	Updated dashboard view.
Pre-condition	Results from the ML model must be received.
Post-condition	The dashboard reflects the updated detection results and metrics.
Dependency	Depends on the successful detection and metrics calculation.
Risk	Failure to update the dashboard can lead to incorrect user perception of the system's status.

3.3 Interface Requirements

3.3.1 User Interfaces

The CodePure extension provides a seamless and user-friendly interface integrated directly into Visual Studio Code (VS Code). The user interface is designed to allow developers to interact intuitively with the code smell detection system while maintaining their development workflow. The following components outline the interface:

GUI

The extension leverages the VS Code Extension API to display results and insights in a structured and visually appealing manner. Key GUI elements include:

- **Tree View Dashboard:** Displays detected code smells and calculated metrics in a hierarchical format within VS Code's Activity Bar. Each file lists various metrics such as LOC (Lines of Code), CBO (Coupling Between Objects), WMC (Weighted Methods per Class), and more.
- **Code Metrics Panel:** Expands individual files to show detailed metric values for each analyzed file, helping developers assess code quality at a glance.
- **Notification System:** Provides real-time pop-up notifications when code analysis is completed, highlighting detected code smells.
- **Status Bar Integration:** Displays the current analysis state (e.g., Ready, Running, or Issues Found) in the VS Code status bar.
- **Code Highlights:** Detected smells are visually marked within the editor, with tooltips explaining the issue and offering potential improvements.

3.3.2 Communications Interfaces

The CodePure extension employs network communication to interact with external services for code smell detection and analysis. The key communication interfaces are as follows:

- **RESTful API Communication:** The extension communicates with a backend through RESTful APIs, which:
  - Send parsed code metrics for analysis.
  - Receive detection results, including identified code smells and suggested refactorings.
- **Data Security:** All communication is encrypted using HTTPS to ensure the confidentiality and integrity of exchanged data. No source code is transmitted—only extracted metrics are sent for processing.
- **Offline Mode:** The extension provides limited functionality in offline mode, allowing developers to view previously calculated metrics and detected smells.

3.3.3 Application Programming Interface (API)

External APIs

CodePure interacts with various external APIs to support its functionality. Below are the key APIs used:

- **VS Code Extension API**

  Purpose: Provides the necessary functions to integrate CodePure as a VS Code extension, enabling custom UI elements, notifications, and event handling.

  Example Usage:

```
vscode.window.createTreeView('codepureMetrics', {...});
```

- **RESTful API for Prediction**

  Purpose: Sends extracted code metrics to the backend service for analysis and code smell detection. The backend processes the data and returns insights.

  Example Usage:

```
response = requests.post("https://api.codepure.com/analyze", json=data)
```

External Libraries

CodePure utilizes several external libraries to support its core functionalities. The following libraries are essential for metric extraction, visualization, and backend communication:

- **JavaParser** (Java)

  Purpose: Parses Java source code to extract structural metrics such as Lines of Code (LOC), Cyclomatic Complexity (CC), Coupling Between Objects (CBO), and Weighted Methods per Class (WMC).

  Usage: Converts Java code into an Abstract Syntax Tree (AST) for metric calculations.

- **VS Code API** (TypeScript)

  Purpose: Creates and manages CodePure's user interface elements, such as the dashboard panel, notifications, and tree views.

  Usage: Used to display code metrics within the VS Code sidebar.

- **Node.js + TypeScript**

  Purpose: Enables efficient development of the CodePure extension, handling asynchronous operations and VS Code API interactions.

  Usage: Implements UI interactions and event-driven processing for real-time metric updates.

3.4 Design Constraints

3.4.1 Standards Compliance

The CodePure extension adheres to the official VS Code extension development guidelines to ensure seamless integration with the IDE. Compliance includes:

1. **Extension Manifest Compliance** – The extension's 'package.json' follows the required VS Code extension manifest structure, defining necessary metadata such as activation events, contributions, and dependencies.

2. **API Usage** – The extension exclusively uses stable VS Code API methods to maintain compatibility with future updates of the IDE.

3. **Security Best Practices** – The extension does not access unnecessary system resources and follows security guidelines to prevent vulnerabilities.

4. **Performance Optimization** – Ensures efficient execution to prevent high CPU/memory usage, maintaining responsiveness within the IDE.

5. **Marketplace Compliance** – The extension package meets the Visual Studio Code Marketplace submission guidelines, ensuring proper licensing, documentation, and user experience expectations.

3.4.2 Software Constraints

The development and deployment of the CodePure extension adhere to specific software constraints, including library and framework versions:

- **VS Code API** – Compatible with VS Code version `1.75.0` or later.

3.4.3 Network Constraints

Users must have a stable internet connection to send code metrics to the web server and receive results promptly, as disruptions may affect usability.

3.5 Non-functional Requirements

- **Performance:** The extension must quickly evaluate code and identify code smells without noticeably degrading performance, even when analyzing multiple files or large projects.

- **Usability:** The user interface must be intuitive and provide actionable feedback on detected code smells, ensuring ease of use for both novice and experienced developers.

- **Reliability:** The extension should gracefully handle invalid inputs without crashing, ensuring consistent operation and dependable results.
- **Scalability:** CodePure must support future expansions, such as additional programming languages, while minimizing the need for major architectural changes. Integration with AWS ensures that the system can scale to accommodate heavy computational workloads if needed.
- **Maintainability:** The codebase must adhere to SOLID principles and clean architecture to facilitate updates, debugging, and enhancements.
- **Security:** CodePure must process all code locally to ensure user privacy and avoid external data transmission.
- **Availability:** The extension must ensure high availability to minimize downtime during updates or heavy usage. Leveraging AWS infrastructure ensures reliability through features like auto-scaling, fault tolerance, and regional redundancy.

3.6 Summary

This chapter defines the key requirements for CodePure, a VS Code extension for detecting code smells. It details Functional Requirements, describing the system's core functionalities such as analyzing code and presenting results. Interface Requirements outline interactions between users and the system, including GUI and API specifications. Design Constraints address limitations like compliance with VS Code guidelines and network dependencies. Lastly, Non-functional Requirements ensure performance, usability, reliability, and security. This chapter provides a structured foundation for the extension's development.

Chapter 4

System Design

4.1 Introduction

This chapter provides an in-depth discussion of the system's architecture and design principles. It outlines the architectural style used, the components and subsystems that constitute the system, and how they interact to achieve the desired functionality. Additionally, this chapter will cover various architectural design viewpoints, including Context Viewpoint, Composition Viewpoint, Logical Viewpoint, Algorithm Viewpoint, and Interaction Viewpoint. It will also discuss Data Design through Dataset Description, Human Interface Design covering User Interface, Screen Objects, and Actions, as well as Implementation, Testing Plan, Requirements Matrix, and System Deployment.

4.2 Design viewpoints

4.2.1 Context viewpoint

The system consists of three main external entities that interact with the core CodePure Extension component. The User interacts with the extension by providing source code for analysis and receiving feedback in the form of detected code smells and calculated code metrics. The Admin entity receives system logs and user feedback for monitoring and maintenance. The system interfaces with a RESTful API hosted on AWS SageMaker,

which processes the extracted code metrics and returns prediction results.

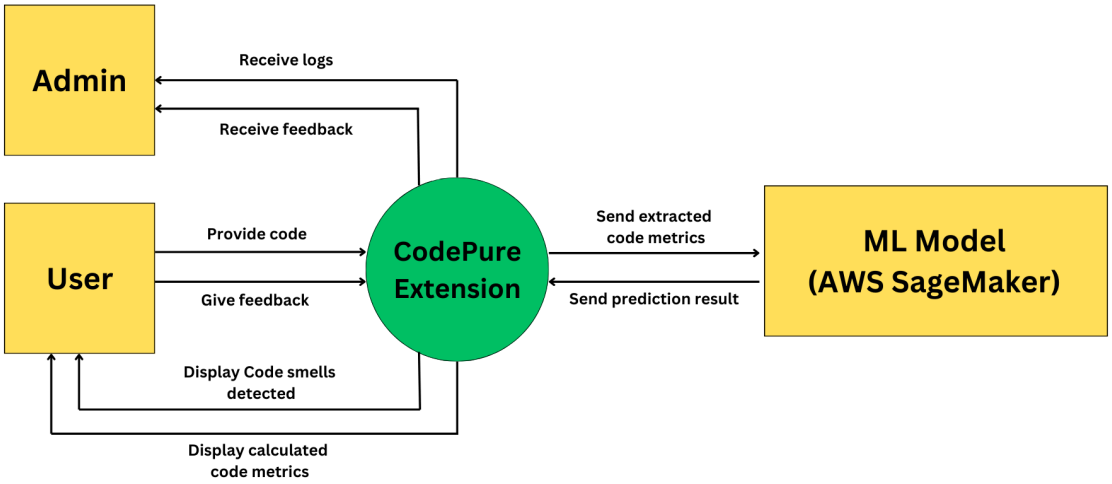


Figure 4.1: Context Diagram

4.2.2 Composition viewpoint

The system employs an Event-Driven Architecture (EDA) to facilitate asynchronous communication and loosely coupled components. This architectural style allows different services to interact efficiently while maintaining independence, which enhances scalability and flexibility. The core principle of EDA is that events are produced by one component and consumed by others, enabling real-time data flow and efficient processing.

Design Entities and Subsystems: The system is decomposed into multiple subsystems, each with a specific role:

- **Event Producer (CodePure Extension):** Generates events (requests) that trigger the system's workflow.
- **Event Router (AWS API Gateway):** Acts as an intermediary, receiving API requests and forwarding them to the appropriate consumer.
- **Event Consumer (AWS Lambda):** Processes the received request, invoking the

required machine learning (ML) model for predictions.

- **Processing Component (AWS SageMaker ML Model):** Performs the necessary computations and returns the prediction results to the Lambda function.
- **Event Broker (AWS SQS):** Serves as a message queue for managing responses from the ML model and ensuring event-driven communication.
- **Event Consumer (CodePure Extension):** Polls the AWS SQS queue to retrieve the processed results and display them to the user.

Responsibilities Partitioning and Subsystem Collaboration:

Each subsystem is assigned specific responsibilities and the system functions through a seamless interaction between these components:

- **CodePure Extension (Producer & Consumer):** Generates API requests and later retrieves the prediction results.
- **AWS API Gateway:** Routes API requests to the Lambda function.
- **AWS Lambda:** Acts as an orchestrator by invoking the ML model, processing responses, and forwarding them to SQS.
- **AWS SageMaker:** Executes the machine learning model to generate predictions.
- **AWS SQS:** Manages the event queue, ensuring reliable message delivery.

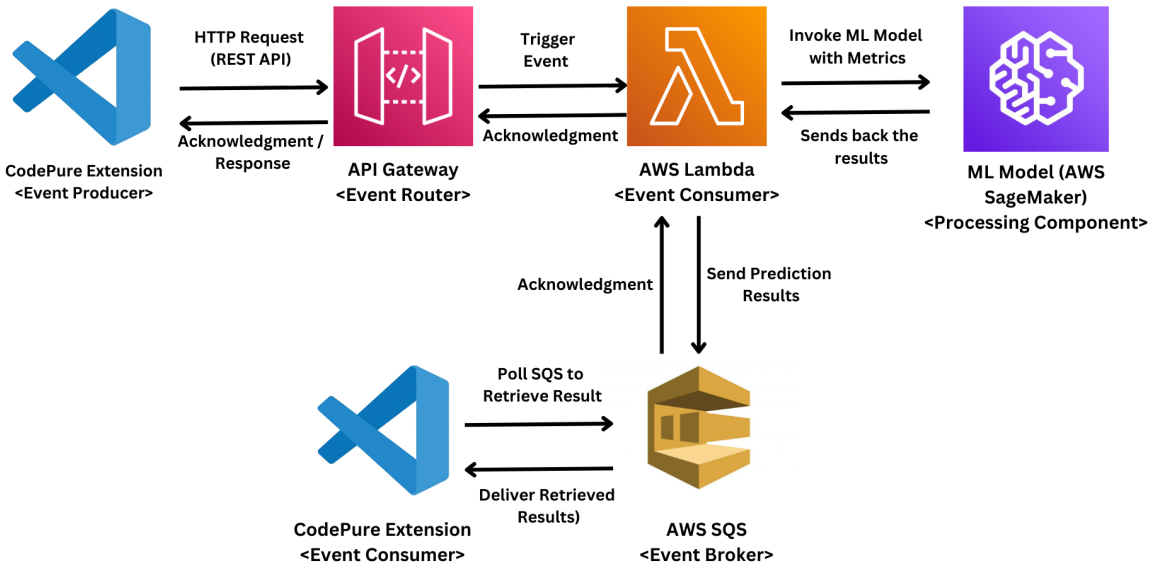


Figure 4.2: Architecture Diagram

The primary reason for selecting an Event-Driven Architecture (EDA) was to achieve a system that is scalable, flexible, and loosely coupled, ensuring seamless integration between independent components. This approach enables efficient asynchronous communication, reducing bottlenecks, improving responsiveness and benefiting from real-time event processing.

4.2.3 Logical viewpoint

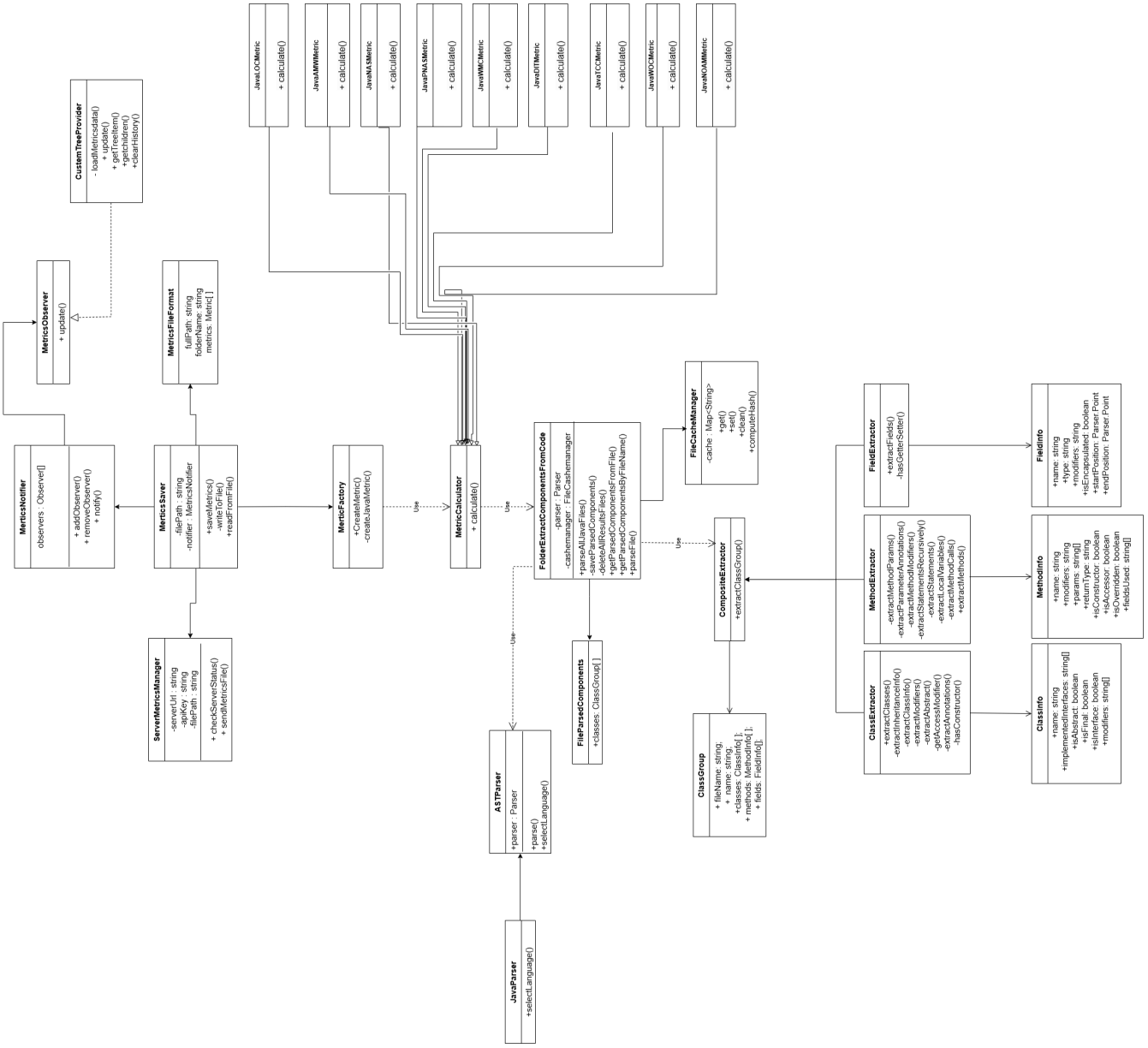


Figure 4.3: Class diagram

Table 4.1: MetricFactory

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	MetricCalculator
Purpose	Dynamically create Java metric object to calculate each metric
Collaborations	MetricCalculator
Attributes	None
Operations	CreateMetric, createJavaMetric,

Table 4.2: MetricCalculator

Abstract or Concrete:	Abstract
Superclasses	MetricFactory
Subclasses	JavaLOCMetric, JavaWOCMetric, JavaDITMetric
Purpose	Abstract class to be implemented by metrics classes to calculate the metrics
Collaborations	JavaParser, ASTParser
Attributes	None
Operations	calculate

Table 4.3: JavaParser

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	ASTParser
Purpose	To select the language in ASTparser to java to begin parsing it
Collaborations	ASTParser
Attributes	None
Operations	selectLanguage

Table 4.4: ASTParser

Abstract or Concrete:	Abstract
Superclasses	JavaParser
Subclasses	None
Purpose	To parse the selected language code
Collaborations	FileParsedComponents
Attributes	None
Operations	selectLanguage, parse

Table 4.5: FileParsedComponents

Abstract or Concrete:	Interface
Superclasses	None
Subclasses	None
Purpose	Contains array that contains the parsed components in the file.
Collaborations	ASTParser
Attributes	classes
Operations	None

Table 4.6: FolderExtractComponentsFromCode

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	Extract components from the whole user's project folder
Collaborations	MetricCalculator, ASTParser, and FileParsedComponents.
Attributes	xxxx
Operations	parseAllJavaFiles, saveParsedComponents, deleteAllResultsFiles, getParsedComponentsFromFile, getParsedComponentsByFileName, parseFile, extractFileComponents, fetchFileContent, parseCode

Table 4.7: JavaLOCMetric

Abstract or Concrete:	Concrete
Superclasses	MetricCalculator
Subclasses	None
Purpose	Calculate number of lines in a class
Collaborations	None
Attributes	None
Operations	calculate

Table 4.8: JavaWOCMetric

Abstract or Concrete:	Concrete
Superclasses	MetricCalculator
Subclasses	None
Purpose	Calculate weight of a class
Collaborations	None
Attributes	None
Operations	calculate, calculateWeight

Table 4.9: JavaDITMetric

Abstract or Concrete:	Concrete
Superclasses	MetricCalculator
Subclasses	None
Purpose	Calculate depth of inheritance
Collaborations	None
Attributes	None
Operations	calculate, findDIT

Table 4.10: ClassGroup

Abstract or Concrete:	Interface
Superclasses	None
Subclasses	None
Purpose	Stores information about grouped components or extracted elements.
Collaborations	CompositeExtractor
Attributes	fileName, name, classes, methods, fields
Operations	None

Table 4.11: CompositeExtractor

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	ClassExtractor, MethodExtractor, FieldExtractor
Purpose	Extracts and organizes multiple components from the source code.
Collaborations	FolderExtractComponentsFromCode, FileCacheManager, and ClassGroup.
Attributes	None
Operations	extractClassGroup

Table 4.12: FileCacheManager

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	Manages caching of parsed files to improve performance.
Collaborations	FolderExtractComponentsFromCode and CompositeExtractor.
Attributes	cache
Operations	get, set and clean

Table 4.13: ClassExtractor

Abstract or Concrete:	Concrete
Superclasses	CompositeExtractor
Subclasses	None
Purpose	Extracts class-related components from source code.
Collaborations	CompositeExtractor and ClassInfo.
Attributes	None
Operations	extractClasses, extractInheritanceInfo, extractClassInfo, extractModifiers, extractAbstract, getAccessModifier, extractAnnotations, isNestedClass, extractGenericParams, hasConstructor

Table 4.14: MethodExtractor

Abstract or Concrete:	Concrete
Superclasses	CompositeExtractor
Subclasses	None
Purpose	Extracts method-related components from source code.
Collaborations	CompositeExtractor and MethodInfo.
Attributes	None
Operations	extractMethodParams, extractParameterAnnotations, extractMethodModifiers, extractStatementsRecursively, extractStatements, extractLocalVariables, extractMethodCalls, extractMethods, isOverriddenMethod, getAccessModifier, extractMethodName, extractMethodReturnType, findParentClass, isConstructorMethod, isAccessor, getFieldsUsedInMethod, extractMethodAnnotations, extractThrowsClause

Table 4.15: FieldExtractor

Abstract or Concrete:	Concrete
Superclasses	CompositeExtractor
Subclasses	None
Purpose	Extracts field-related components from source code.
Collaborations	CompositeExtractor and FieldInfo.
Attributes	None
Operations	extractFields, hasGetterSetter,

Table 4.16: ClasssInfo

Abstract or Concrete:	Interface
Superclasses	None
Subclasses	None
Purpose	Gather information from a class
Collaborations	ClassExtractor
Attributes	name, implementedInterfaces, isAbstract, isFinal, isInterface, modifiers, annotations, startPosition, endPosition, isNested, genericParams, hasConstructor, parent
Operations	None

Table 4.17: MethodInfo

Abstract or Concrete:	Interface
Superclasses	None
Subclasses	None
Purpose	Gather method's information from a class
Collaborations	MethodExtractor
Attributes	name, modifiers, params, returnType, isConstructor, isAccessor, isOverridden, fieldsUsed, annotations, throwsClause, methodBody, localVariables, methodCalls, startPosition, endPosition
Operations	None

Table 4.18: FieldInfo

Abstract or Concrete:	Interface
Superclasses	None
Subclasses	None
Purpose	Gather fields information from a class
Collaborations	FieldExtractor
Attributes	name, type, modifiers, isEncapsulated, startPosition, endPosition
Operations	None

Table 4.19: MetricsSaver

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	Save metrics in Json file
Collaborations	MetricFactory, MetricsFileFormat, MetricsNotifier, ServerMetricsManager
Attributes	filePath, notifier
Operations	saveMetrics, writeToFile

Table 4.20: ServerMetricsManager

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	Send metrics calculated to the server
Collaborations	MetricsSaver
Attributes	serverUrl, apiKey, filePath
Operations	checkServerStatus, sendMetricsFile,

Table 4.21: MetricsFileFormat

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	Interface that structure metrics in json
Collaborations	MetricsSaver
Attributes	fullPath, folderName, metrics
Operations	None

Table 4.22: MetricsNotifier

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	When changes occur it notify all the observers
Collaborations	MetricsSaver
Attributes	observers
Operations	addObserver, removeObserver, notify

Table 4.23: MetricsObserver

Abstract or Concrete:	Interface
Superclasses	None
Subclasses	None
Purpose	Update the dashboard
Collaborations	MetricsNotifier
Attributes	None
Operations	update

Table 4.24: CustomTreeProvider

Abstract or Concrete:	Concrete
Superclasses	None
Subclasses	None
Purpose	implements the observers
Collaborations	MetricsObserver
Attributes	None
Operations	loadMetricsData, update, getTreeItem, getChildren, clearHistory,

4.2.4 Patterns use viewpoint

The Observer and Factory Method patterns are employed to enhance modularity, maintainability, and scalability. The Observer pattern is applied in the MetricObserver and MetricNotifier classes. This pattern allows multiple metric observers to subscribe to updates from the metric notifier, ensuring that any changes in metric data are efficiently communicated. This approach enhances modularity by decoupling the data processing logic from the notification mechanism, making the system more extensible for future metric observers without modifying existing code.

The Factory Method pattern is utilized in the MetricFactory class to create metric instances dynamically. By employing this pattern, the system provides a flexible way to instantiate different types of metrics (JavaLOCMetric, JavaNOMMetric, etc.) without directly specifying their concrete implementations. This design promotes scalability, as new metric types can be introduced by extending the factory without altering client code. By leveraging these patterns, the system achieves a more structured and reusable architecture, allowing seamless integration of new features while maintaining clean code separation.

4.2.5 Algorithm viewpoint

The dataset is first pre-processed using the Synthetic Minority Over-sampling Technique (SMOTE) to address class imbalance in the multilabel dataset. SMOTE generates synthetic samples for the minority classes to create a more balanced dataset. After balancing the data, a hybrid feature selection approach is applied, which combines Filter and Em-

bedded methods. In the Filter method, feature importance is assessed using the Receiver Operating Characteristic (ROC) curve, selecting relevant features based on their ability to discriminate between classes. The Embedded method uses Lasso (Least Absolute Shrinkage and Selection Operator) for feature selection by applying L1 regularization to shrink less important features, enhancing model interpretability and reducing overfitting. Following data preprocessing and feature selection, multiple machine learning models (Random Forest, Gradient Boosting, XGBoost, Logistic Regression, and K-Nearest Neighbors) are trained on the processed dataset. These models are implemented within a MultiOutputClassifier framework to handle the multilabel classification.

4.2.6 Interaction viewpoint

This sequence diagram illustrates the workflow of the CodePure VS Code extension for detecting code smells. The process begins when the user writes and saves code, prompting the extension to parse the code and generate relevant metrics. These metrics are then sent through AWS Gateway and forwarded to AWS Lambda. The Lambda function invokes an ML model hosted on AWS SageMaker, which analyzes the metrics to detect potential code smells. The results are then sent back through the pipeline, ultimately displaying the findings to the user in the VS Code extension.

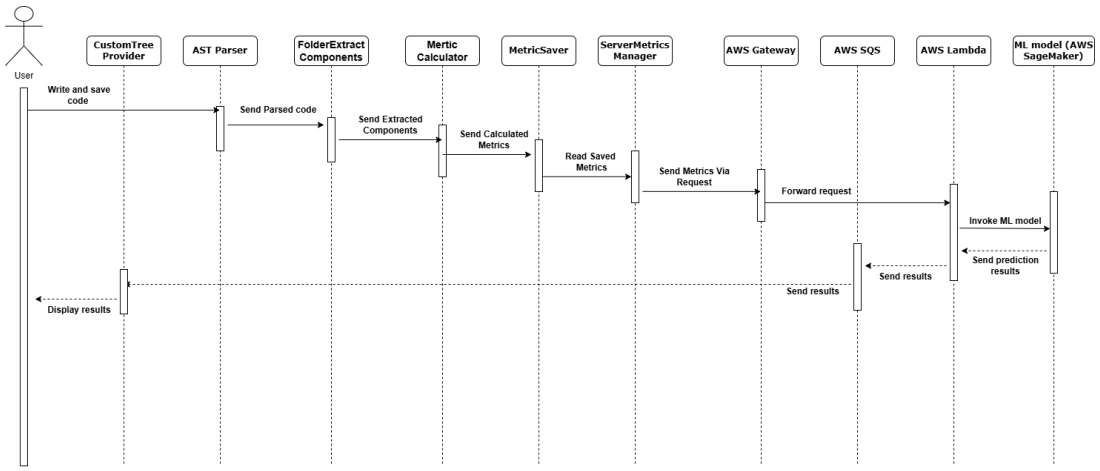


Figure 4.4: System Sequence Diagram

The user sequence diagram represents a process where a user writes and saves code, which is then processed through multiple stages. The CustomTree Provider parses the code and

sends it to the AST Parser, which further processes and forwards it to FolderExtract Components. Extracted components are sent to the Metric Calculator, which computes metrics and forwards them to the ML Model for prediction. The results are then sent back to the user for display.

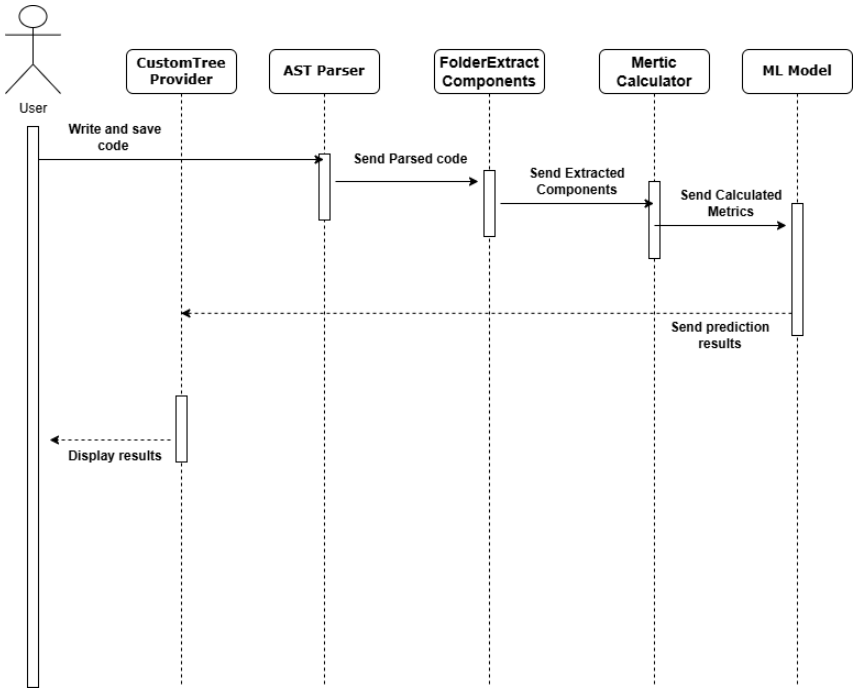


Figure 4.5: User Sequence Diagram

The AWS sequence diagram illustrates the flow of metrics from the CodePure Extension to an AWS-based system. The extension sends a request to AWS Gateway, which forwards it to AWS SQS. From there, AWS Lambda invokes an ML model hosted on AWS SageMaker. The model processes the data, generates predictions, and stores the results in Amazon DynamoDB. Finally, the results are sent back through the system to be displayed in the CodePure Extension.

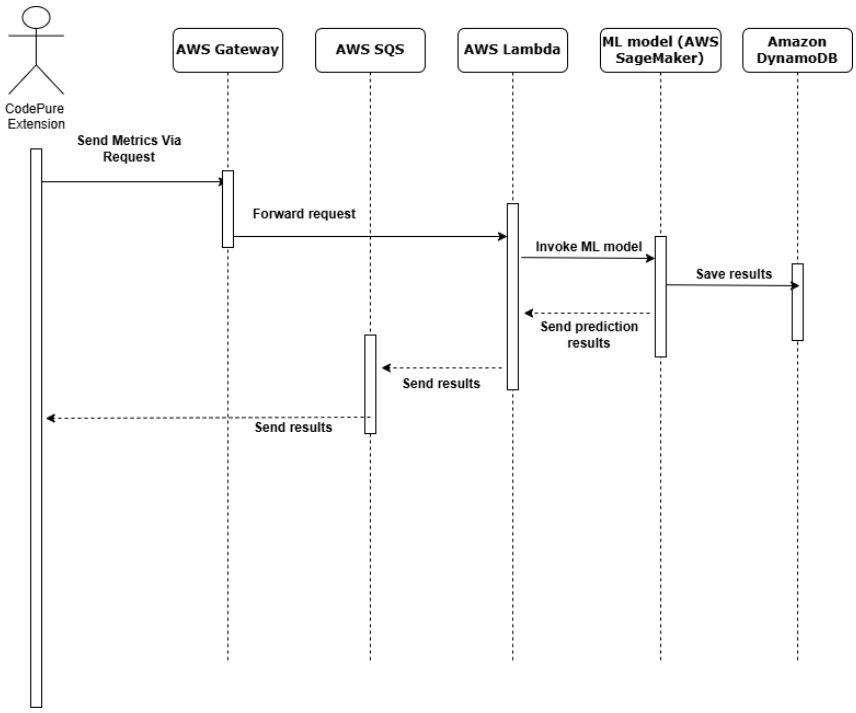


Figure 4.6: AWS Sequence Diagram

4.3 Data Design

4.3.1 Dataset Description

Our project utilizes a multi-label dataset designed for detecting code smells. This datasets is crucial for training and evaluating machine learning models aimed at identifying problematic code structures that may affect maintainability and readability.

The dataset is concerned with classes, each annotated with multiple code smell labels. These labels indicate the presence of different code smells such as "God Class," "Brain Class," "Data Class," and others. Additionally, the dataset contains several software metrics, including Weighted Methods per Class (WMC), Tight Class Cohesion (TCC), and other attributes.

Dataset Name	Class Level Dataset
Link	`https://figshare.com/articles/conference_contribution/mlCodesmell/21343299/2?file=37883547`
Size	373401 rows
Number of Labels	8
Number of Features	41
Format	CSV file
File Size	70 MB
Notes	The dataset is a multi-label source code metric dataset. It is helpful to build machine learning models to detect code-smell based on multi-label classification methods

Table 4.25: Class Level Dataset Details

4.3.2 Database Design

Our MongoDB database is a NoSQL database designed to support a system for detecting and managing code smells within software repositories. The database store and organize metadata related to

- Source files and the code smells they contain.
- Predictions and analysis history.
- Repository and user data.
- Feedback from users.

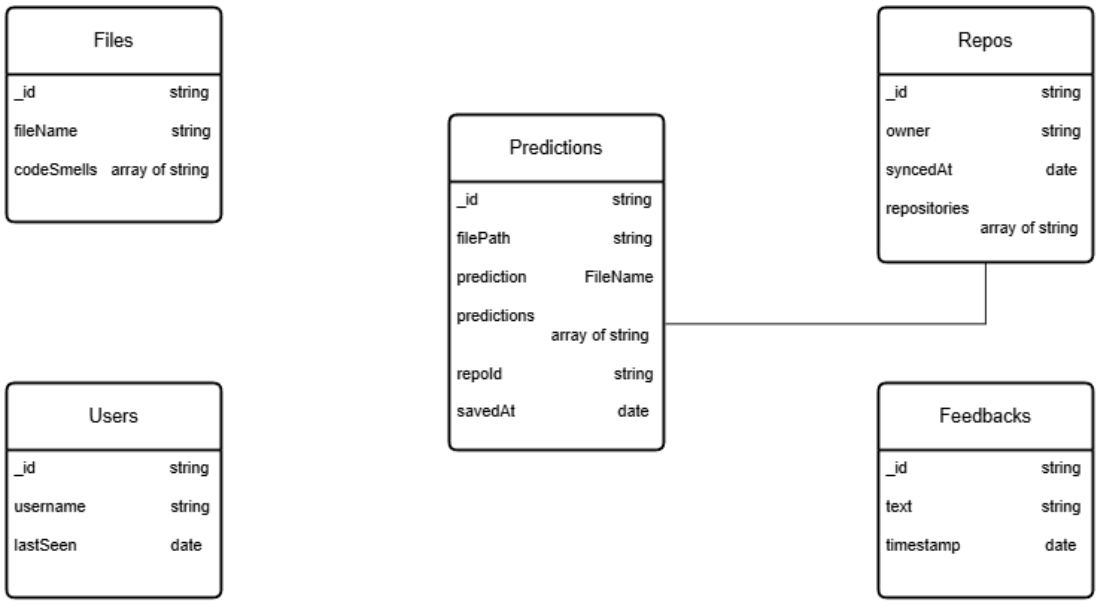


Figure 4.7: Database schema

4.4 Human Interface Design

4.4.1 User Interface

The user interface of the CodePure extension is designed to be intuitive and seamlessly integrated into the development environment. From the user's perspective, the system provides the following functionalities:

1. **Error Validation**: The system validates the user's code for errors and ensures that only supported languages are analyzed.

2. **Code Analysis**: Users can trigger an analysis of their code, which will be parsed and evaluated for potential issues, such as code smells and showing metrics.

3. **Real-Time Feedback**: The system provides immediate feedback on detected issues, displaying relevant insights and suggested improvements directly within the IDE in the extension's dashboard.

4. **Dashboard Visualization**: Users can access a dashboard that summarizes the calculated metrics, historical trends, and overall code quality assessment.

5. **User Feedback**: Users can submit feedback on detected issues, helping refine the analysis and improve the accuracy of the system.

The user experience is designed to be seamless, with minimal manual intervention required. The integration within the development environment ensures that users can continuously improve their code quality without disrupting their workflow.

4.4.2 Screen Images

1. **Error Validation**:

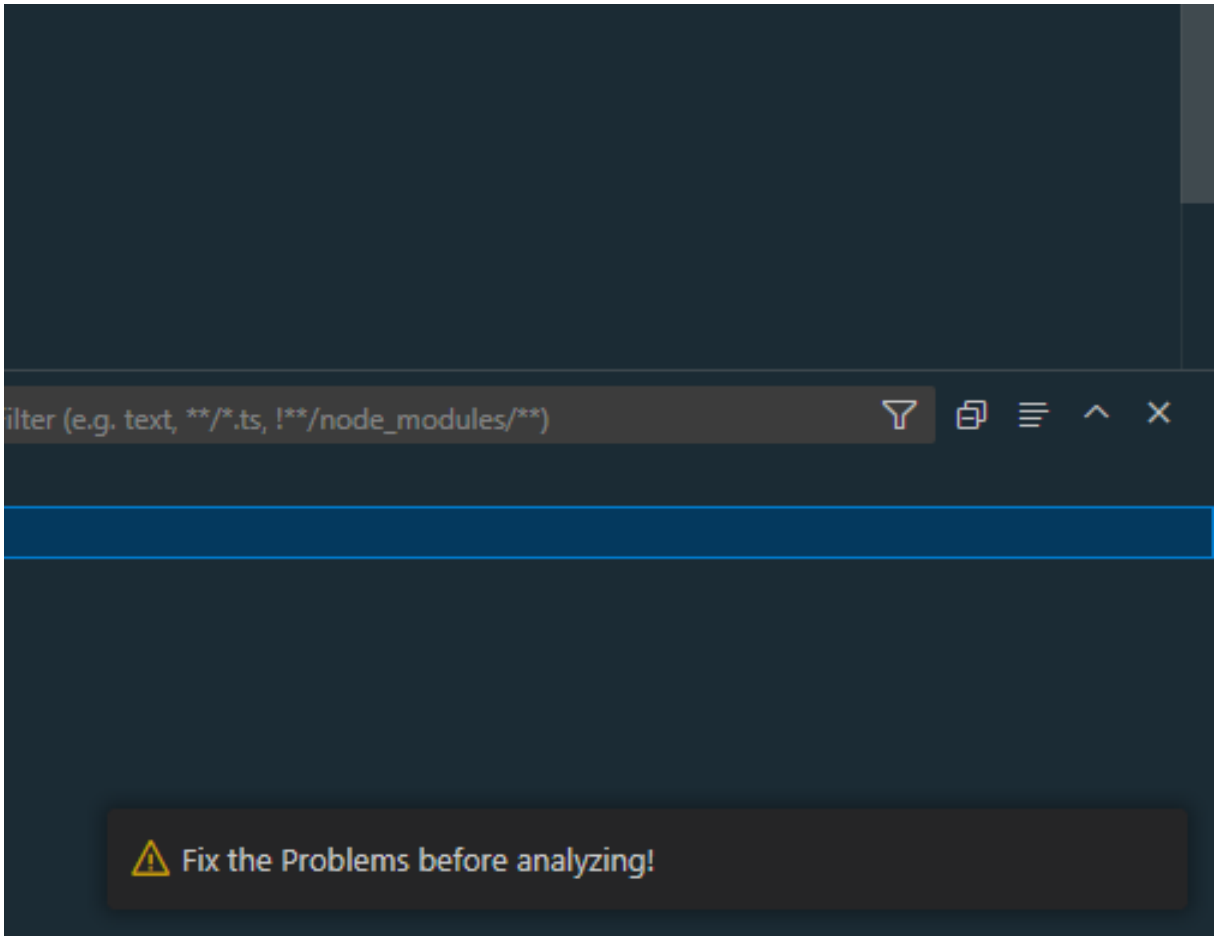


Figure 4.8: Fix the error in code

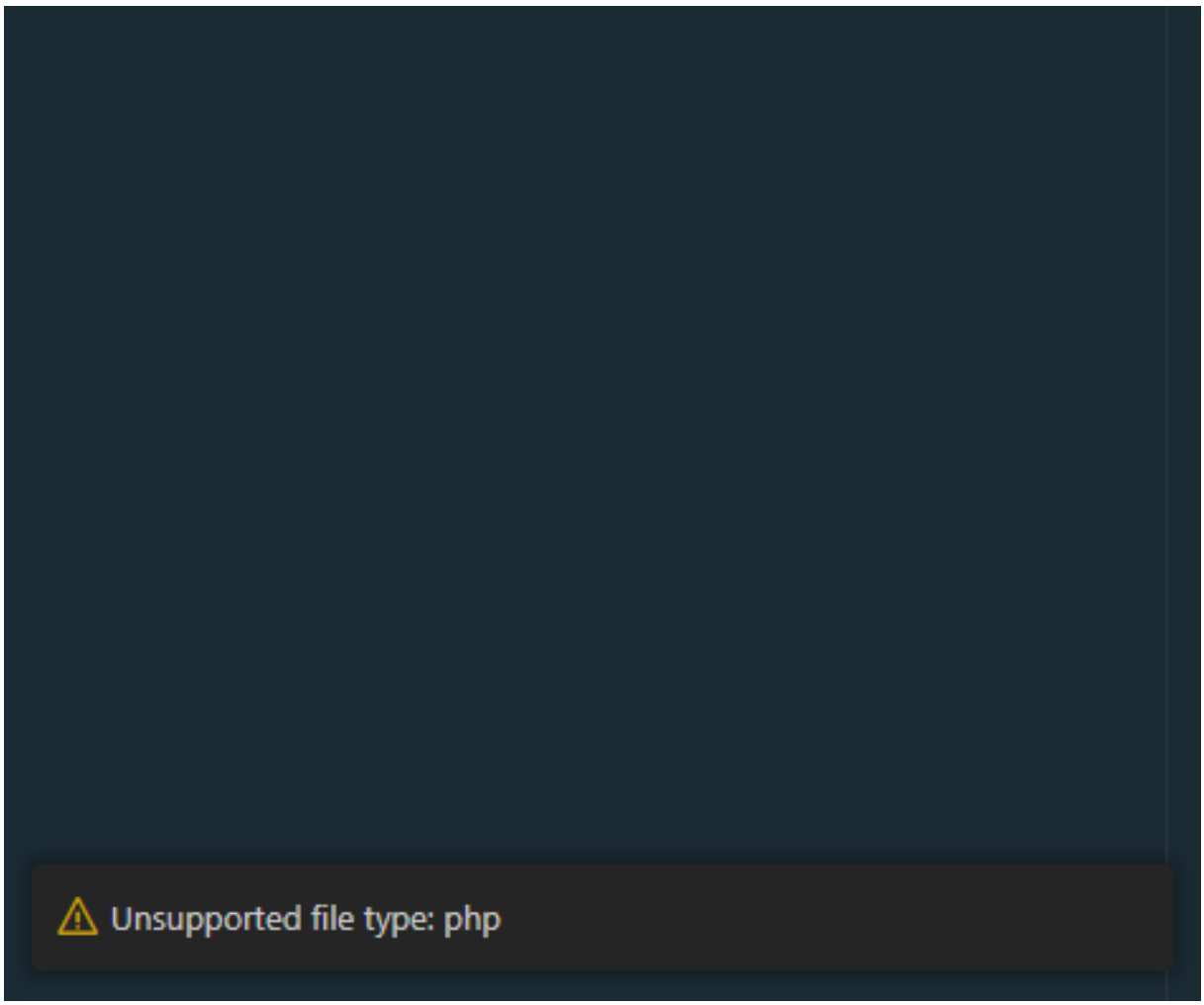


Figure 4.9: unsupported language

2. **Code Analysis**: The extension will show a message to press " Ctrl + S " to begin the analyze.

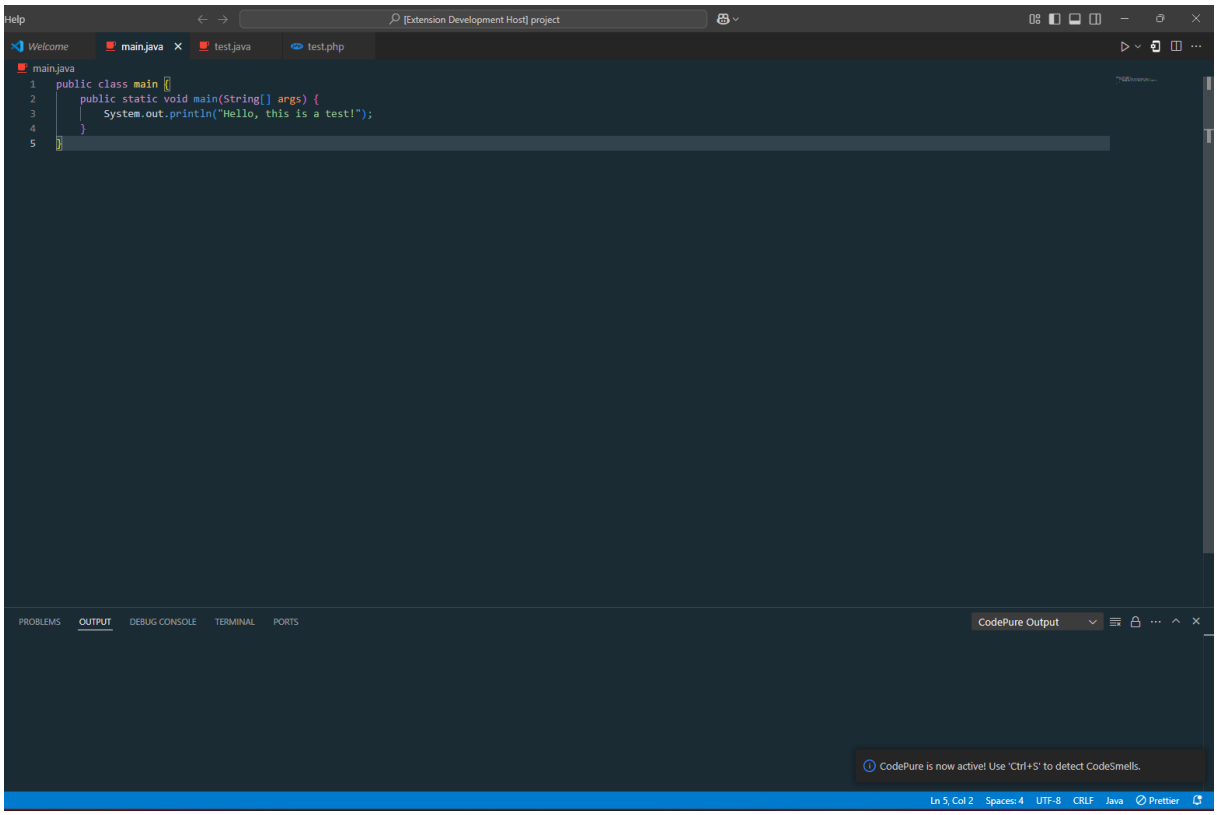


Figure 4.10: ready to analyze the code

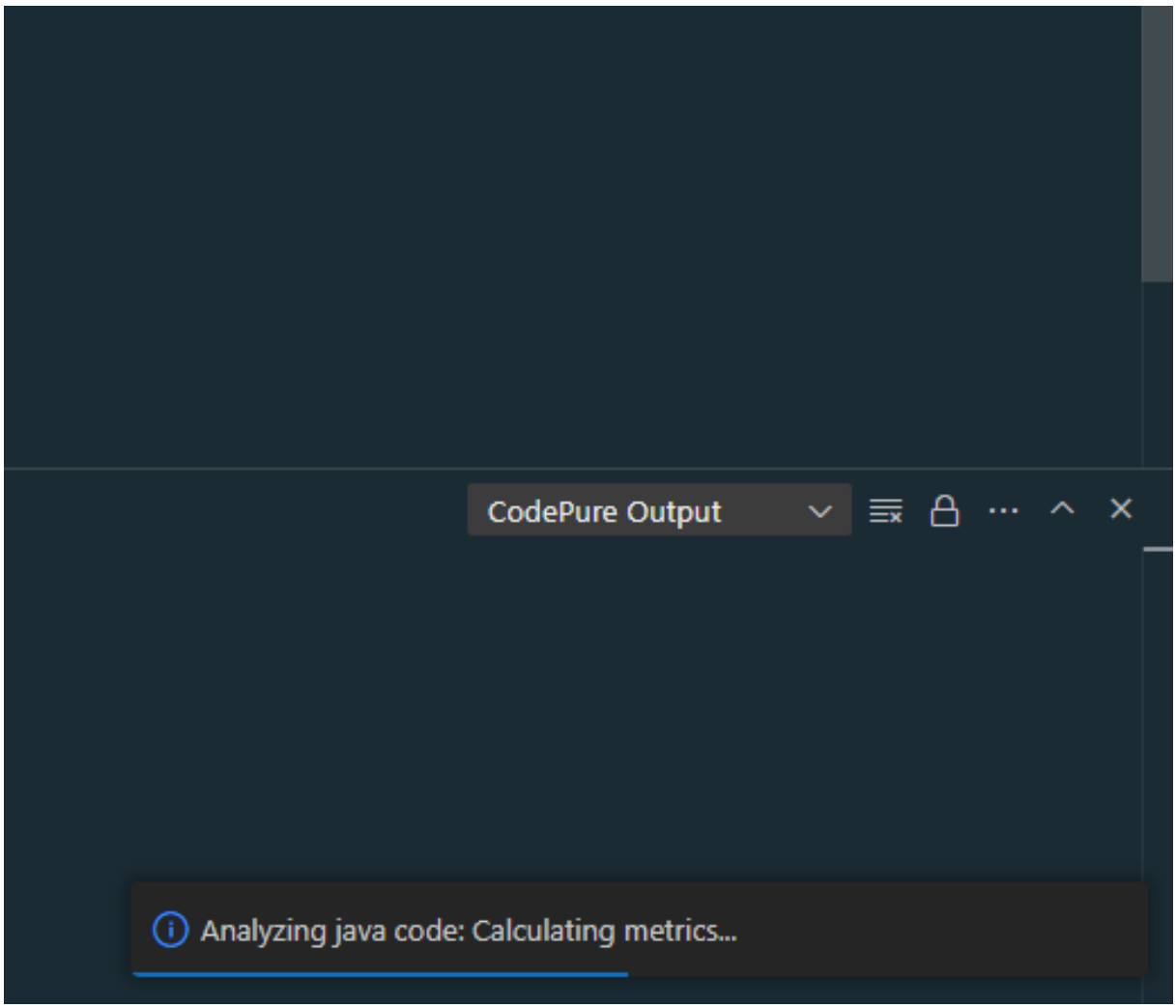


Figure 4.11: analyzing the file

3. **Code Parsing and Extracting Components**: Extracting components from the file and save it in a JSON file.


```
"name": "CanalInstanceWithSpring",
"classes": [
  {
    "name": "CanalInstanceWithSpring",
    "implementedInterfaces": [],
    "isAbstract": false,
    "isFinal": false,
    "isInterface": false,
    "AccessLevel": "public",
    "annotations": [],
    "startPosition": {
      "row": 24,
      "column": 67
    },
    "endPosition": {
      "row": 63,
      "column": 1
    },
    "parent": "AbstractCanalInstance",
    "isNested": false,
    "hasConstructor": false
  }
```


Figure 4.12: JSON file with extracted components

```
"methods": [
  {
    "name": "start",
    "modifiers": "public",
    "params": [],
    "returnType": "Object",
    "isConstructor": false,
    "isAccessor": false,
    "isOverridden": false,
    "fieldsUsed": [],
    "annotations": [],
    "throwsClause": [],
    "methodBody": [
      "expression_statement",
      "expression_statement"
    ],
    "localVariables": [],
    "methodCalls": [
      "logger.info",
      "super.start"
    ],
    "fieldAccess": [],
    "parent": null,
    "startPosition": {
      "row": 28,
      "column": 4
    },
    "endPosition": {
      "row": 31,
      "column": 5
    }
  },
  {
    "name": "setDestination",
    "modifiers": "public",
    "params": [
      " String destination"
    ],
    "returnType": "String",
    "isConstructor": false,
```


Figure 4.13: JSON file with extracted components

4. **Calculating metrics**: Calculating metrics using the extracted components.

```
export class JavaWeightedMethodCount extends MetricCalculator {

    Codeium: Refactor | Explain | Generate JSDoc | ×
    calculate(node: any, FECFC: FolderExtractComponentsFromCode, Filename: string): number {
        let allClasses: ClassInfo[] = [];
        let allMethods: MethodInfo[] = [];
        let allFields: FieldInfo[] = [];

        const fileParsedComponents = FECFC.getParsedComponentsByFileName(Filename);

        if (fileParsedComponents) {
            const classGroups = fileParsedComponents.classes;
            classGroups.forEach((classGroup) => {
                allClasses = allClasses.concat(classGroup.classes);
                allMethods = allMethods.concat(classGroup.methods);
                allFields = allFields.concat(classGroup.fields);
            });
        }

        return this.calculateWeightedMethodCount(allMethods);
    }

    Codeium: Refactor | Explain | Generate JSDoc | ×
    private calculateWeightedMethodCount(methods: MethodInfo[]): number {
        let complexity = 0;

        methods.forEach((method) => {
            if (method.methodBody.length > 0) {
                complexity++;

                method.methodBody.forEach((statement) => {
                    if (
                        statement === "if_statement" ||
                        statement === "while_statement" ||
                        statement === "do_statement" ||
                        statement === "case" ||
                        statement === "for_statement" ||
                        statement === "condition" ||
                        statement === "ternary_expression"
                    ) {
                        complexity++;
                    }
                });

            }
        });

        return complexity;
    }
```

Figure 4.14: Calculating metrics

```
export class JavaNumberOfAccessorMethods extends MetricCalculator {
  Codeium: Refactor | Explain | Generate JSDoc | X
  calculate(node: any,  FECFC: FolderExtractComponentsFromCode, Filename: string): number
  {
    let allClasses: ClassInfo[] = [];
    let allMethods: MethodInfo[] = [];
    let allFields: FieldInfo[] = [];

    const fileParsedComponents = FECFC.getParsedComponentsByFileName(Filename);

    if (fileParsedComponents)
      {
        const classGroups = fileParsedComponents.classes;
        classGroups.forEach((classGroup) =>
        {
          allClasses = allClasses.concat(classGroup.classes);
          allMethods = allMethods.concat(classGroup.methods);
          allFields = allFields.concat(classGroup.fields);
        });
      }

    const NOAM = this.findaccessedmethods(allMethods);

    return NOAM;
  }

  Codeium: Refactor | Explain | Generate JSDoc | X
  private findaccessedmethods(Methods: MethodInfo[]): number {
    let NOAM = 0;

    for (const Method of Methods) {
      if (Method.isAccessor) {
        NOAM++;
      }
    }

    return NOAM;       zeyad zayaty, 4 days ago • #Delete all the file extracted after clos
  }
```

Figure 4.15: Calculating metrics

5. **Saving metrics**: Saving metrics in JSON file.


```
"folderName": "spring/CanalInstanceWithSpring.java",
"metrics": [
  {
    "name": "LOC",
    "value": 40
  },
  {
    "name": "AMW",
    "value": 1
  },
  {
    "name": "CBO",
    "value": 1
  },
  {
    "name": "ATFD",
    "value": 0
  },
  {
    "name": "FDP",
    "value": 1
  },
  {
    "name": "DAC",
    "value": 1
  },
  {
    "name": "WMC",
    "value": 8
  },
  {
    "name": "WOC",
    "value": 0.125
```


Figure 4.16: Saving metrics

```
async sendMetricsFile() {
  let filePath = path.join(
    __dirname,
    "..",
    "src",
    "Results",
    "MetricsCalculated.json"
  );

  filePath = filePath.replace(/out[\\\/]?/, "");

  try {
    const fileContent = fs.readFileSync(filePath, 'utf8');
    if (!fileContent.trim()) {
      console.log('The metrics file is empty.');
      return;
    }

    const metricsData = JSON.parse(fileContent);

    const response = await fetch(`${this.serverUrl}/metrics`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'x-api-key': this.apiKey,
      },
      body: JSON.stringify(metricsData),
    });

    if (response.ok) {
      console.log('CodePure Extension: Metrics Sent To The Server.');
      const data = await response.json();
      console.log(data);
      return data;
    } else {
      console.error('Failed to send metrics file:', response.statusText);
      return null;
    }
  } catch (error) {
    console.error('Error connecting to server:', error);
    return null;
  }
}
```

Figure 4.17: Saving metrics

6. **Receiving metrics and sending back results**: receiving metrics from extension.

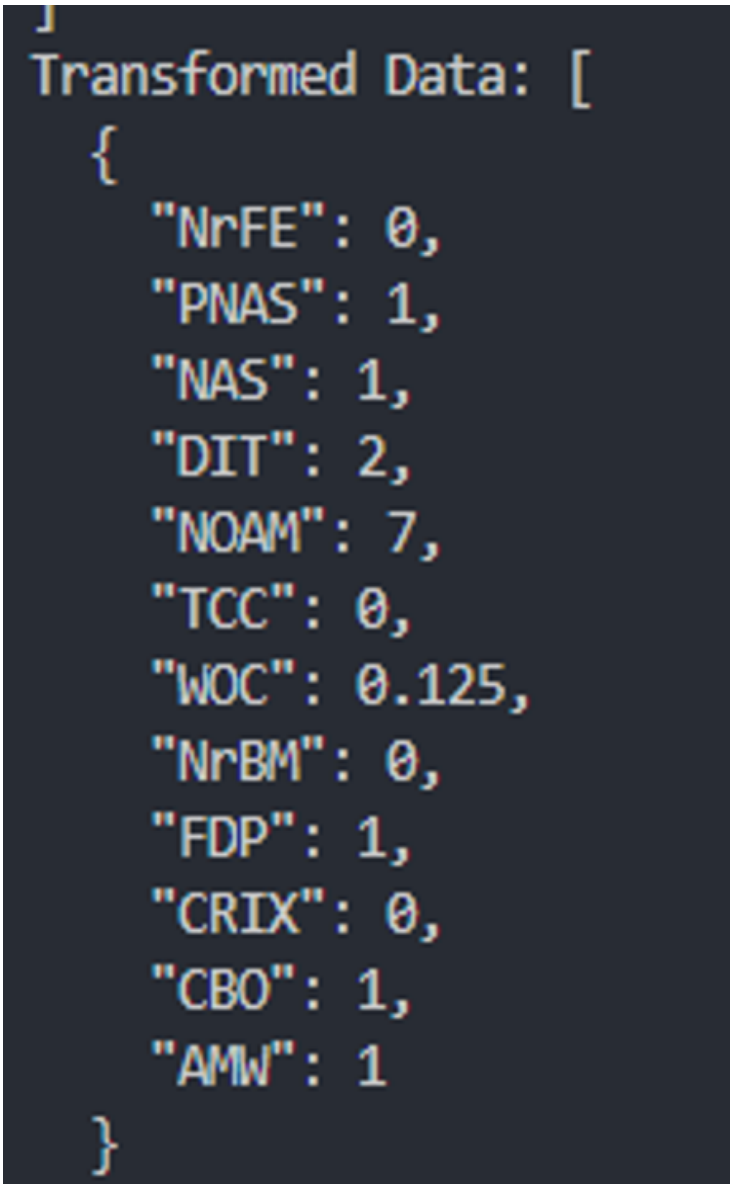


Figure 4.18: receive metrics

```
app.post("/metrics", async (req, res) => {
  const metrics = req.body;


  if (!Array.isArray(metrics) || metrics.length === 0) {
    return res.status(400).json({
      error: "Invalid metrics data: expected an array with metrics objects.",
    });
  }
  const validMetric = metrics.every(
    (item) =>
      item.hasOwnProperty("fullPath") &&
      item.hasOwnProperty("folderName") &&
      Array.isArray(item.metrics) &&
      item.metrics.every(
        (metric) =>
          metric.hasOwnProperty("name") && metric.hasOwnProperty("value")
      )
  );

  if (!validMetric) {
    return res.status(400).json({
      error:
        'Invalid metric format: each item should have "fullPath", "folderName", and "metrics" with valid "name" and "value".',
    });
  }

  try {

    console.log("Received Metrics:", JSON.stringify(metrics, null, 2));
      const predictions = await makePrediction(metrics);
    res.json({       You, 6 days ago • KNN model prediction
      message: "CodePure server: data received successfully.",
      predictions: predictions,
    });
  } catch (error) {
    console.error("Error processing metrics and predictions:", error);
    res.status(500).json({
      error: "Error processing metrics and making predictions",
      details: error.message,
    });
  }
});
```

Figure 4.19: send results

7. **Receiving prediction results**: receiving results from ML model from the server.

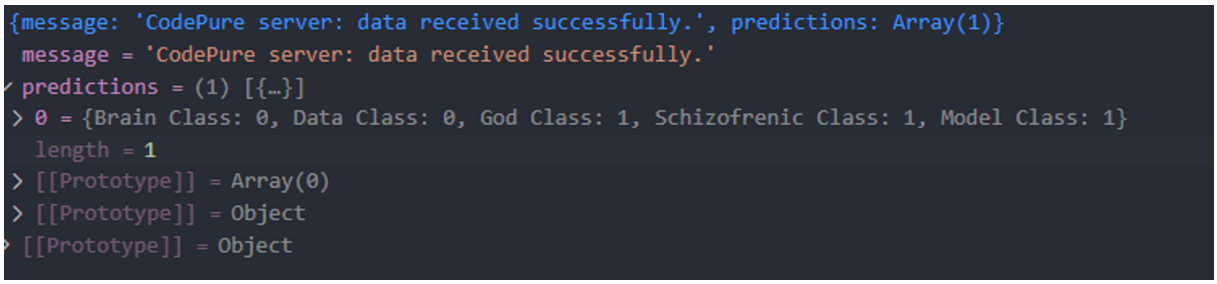


Figure 4.20: receive metrics

8. **Real-Time Feedback**: Once the user press " Ctrl + S " the extension will show the metrics calculated and the code smells in the dashboard.

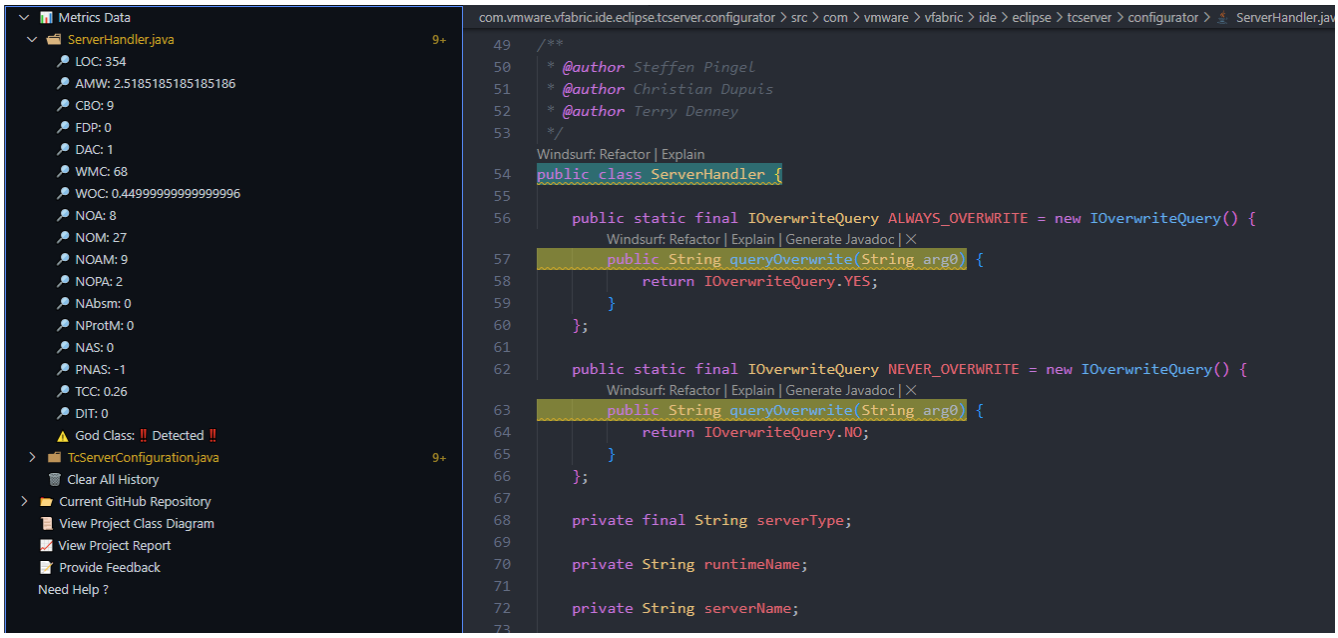


Figure 4.21: Real-Time Feedback

9. **Dashboard Visualization**: User can access every file in his project showing its metrics and the code smells detected.

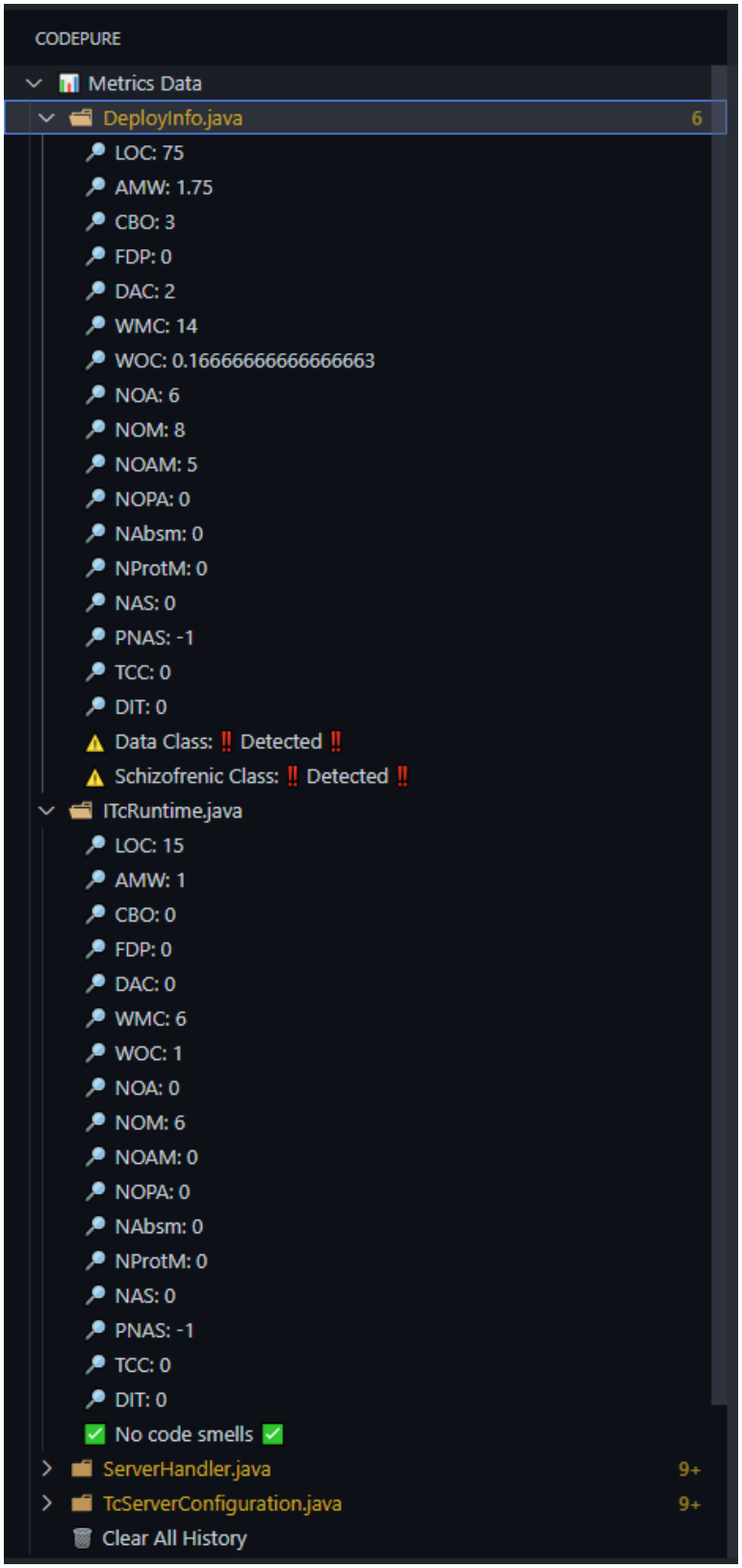


Figure 4.22: Dashboard

10. **User Feedback**: At the end, User can give a feedback on the results provided by the extension.

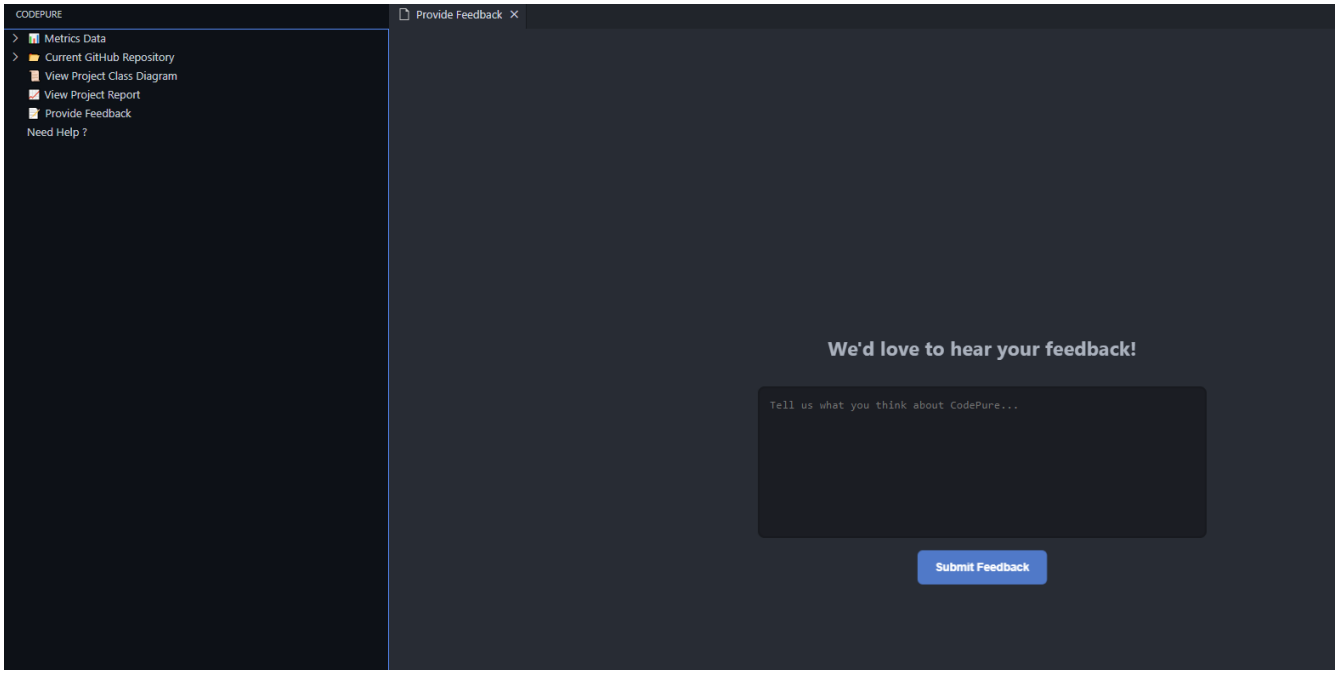
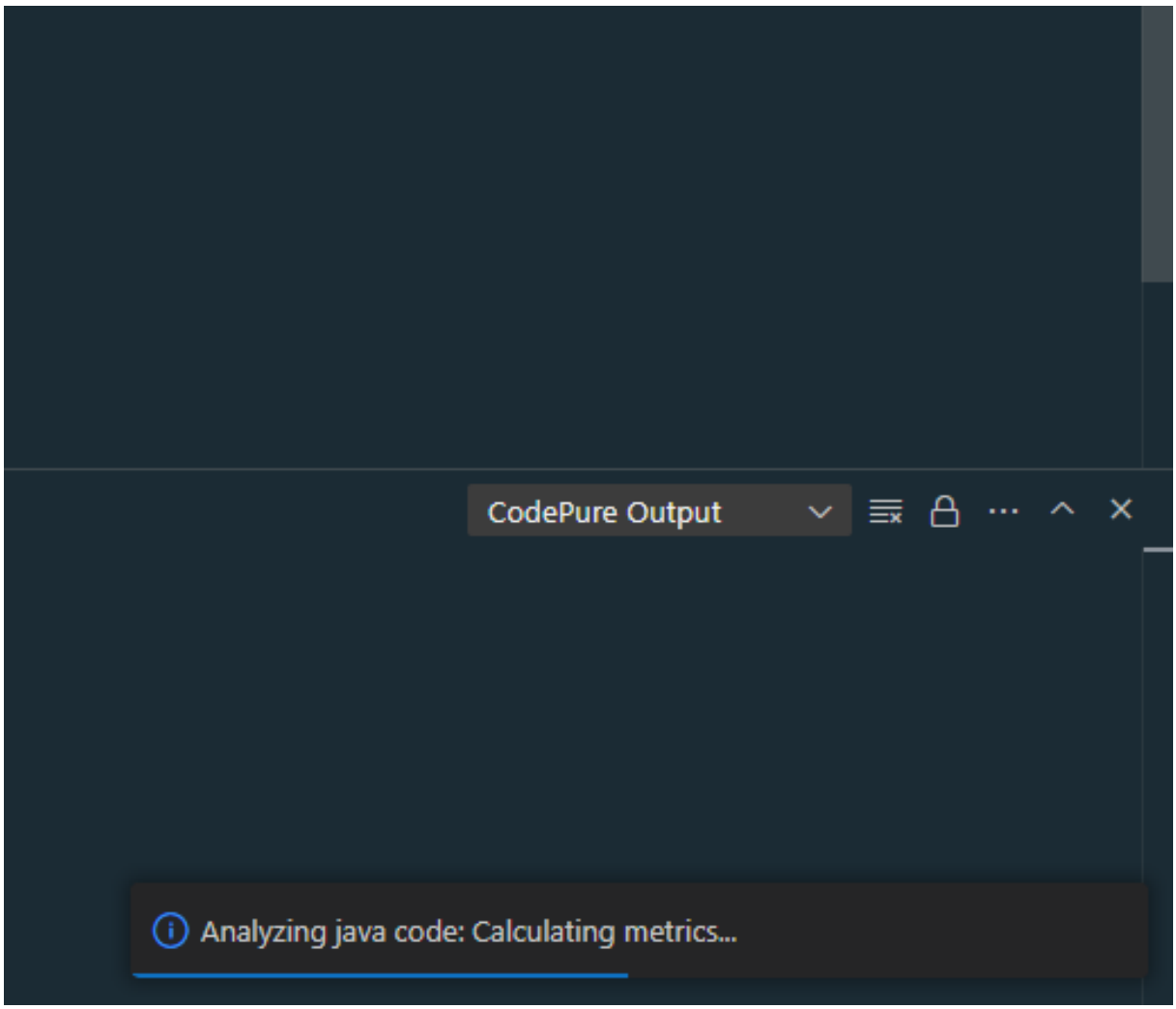


Figure 4.23: Feedback

4.4.3 Screen Objects and Actions

- Once the user press " Ctrl + S " the extension will start to analyze the code and show the metrics calculated and the code smells (not implemented yet) in the dashboard.



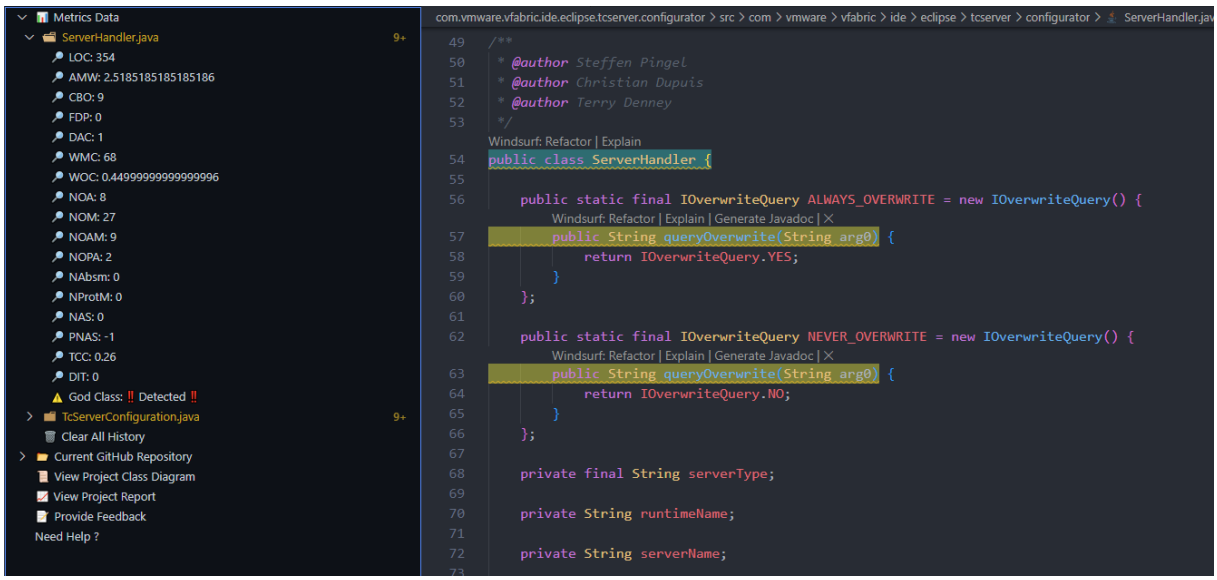


Figure 4.24: "Ctrl + S " action

4.5 Implementation

This section provides a detailed description of the implementation of CodePure, a Visual Studio Code extension developed to detect code smells. The system is implemented using TypeScript for the extension and Python for the machine learning model. It integrates various libraries and services to ensure efficient parsing, processing, and detection of code smells. Communication between components is managed via AWS services to enable seamless interaction between the extension and the machine learning model.

4.5.1 Programming Languages

- **TypeScript**: Used for implementing the VS Code extension, ensuring type safety and maintainability.
- **Python**: Used for the machine learning model responsible for analyzing extracted software metrics and detecting code smells.

4.5.2 Libraries and Frameworks

- **Tree-sitter**: A parsing library used in the VS Code extension to analyze code structure and parse it and help to extract essential software metrics.
- **AWS SDK**: Used in the extension to facilitate communication with AWS services.
- **NumPy & Pandas**: Used in the machine learning model for data processing.
- **Scikit-learn & imbalanced-learn**: Utilized for training and running the ML model, handling imbalanced data and feature selection

4.5.3 Platforms and Services

- **Visual Studio Code**: The development environment for the extension.
- **AWS API Gateway**: Serves as the communication bridge between the extension and backend services.
- **AWS Lambda**: Handles the processing of requests and invokes the machine learning model.
- **AWS SageMaker**: Hosts and deploys the machine learning model for scalable inference.

4.6 Testing Plan

4.6.1 Test Scenario 1: Unsupported Language Detection

Description: The user attempts to use the extension on a file written in an unsupported programming language. The system should detect that the language is not supported and display an appropriate error message without proceeding with analysis.

Test Cases

Table 4.26: Test Scenario: Unsupported Language Detection

Test Case ID	TC01
Description	The user attempts to analyze a file in an unsupported programming language.
Functional Req Code	FR06
Test Data	A file with a ‘.php‘ extension (if only Java and another language will be decided later are supported).
Expected Result	The system displays an error message: "Unsupported language." No analysis is performed.

4.6.2 Test Scenario 2: File Contains Syntax Errors

Description: The user attempts to use the extension on a file that contains syntax errors. The system should detect the errors and prevent further analysis.

Test Cases

Table 4.27: Test Scenario: File Contains Errors

Test Case ID	TC02
Description	The user attempts to analyze a file that contains syntax errors.
Functional Req Code	FR06
Test Data	A ‘.java‘ file with syntax errors (e.g., missing semicolons, unmatched brackets).
Expected Result	The system displays an error message: "Code contains errors. Please fix them before analysis." No code analysis is performed.

4.6.3 Test Scenario 3: Successful Metrics Calculation

Description: The user analyzes a correctly formatted file, and the system successfully calculates and displays the required metrics.

Test Cases

Table 4.28: Test Scenario: Successful Metrics Calculation

Test Case ID	TC03
Description	The user analyzes a valid file, and the system calculates and displays the results.
Functional Req Code	FR07
Test Data	A well-formed '.java' file with multiple classes and methods.
Expected Result	The system successfully calculates and displays metrics such as TCC.

4.6.4 Test Scenario 4: Handling Large Codebases

Description: The user analyzes a project with multiple large files, and the system must process them efficiently without crashing.

Test Cases

Table 4.29: Test Scenario: Handling Large Codebases

Test Case ID	TC04
Description	The user analyzes a project containing multiple '.java' files with thousands of lines of code.
Functional Req Code	FR06
Test Data	A project with at least 10 large '.java' files.
Expected Result	The system successfully processes all files without performance degradation or crashing.

4.6.5 Test Scenario 5: AI Quick Fix for Code Smells

Description: When a code smell is detected, the system suggests a fix using Gemini AI.

Test Cases

Table 4.30: Test Scenario: AI Quick Fix for Code Smells

Test Case ID	TC05
Description	The system suggests an AI-generated fix when a code smell is detected.
Functional Req Code	-
Test Data	A ‘.java‘ file containing code with a known code smell (e.g., brain class).
Expected Result	The system detects the code smell and provides an AI-generated fix suggestion.

4.6.6 Test Scenario 6: GitHub Sign-In Requirement for Viewing Metrics & Repositories

Description: Users must sign in to GitHub to access calculated metrics, detected code smells, and their repositories.

Test Cases

Table 4.31: Test Scenario: GitHub Sign-In Requirement

Test Case ID	TC06
Description	The user must sign in to GitHub to view metrics, detected code smells, and repositories.
Functional Req Code	-
Test Data	A user trying to access the metrics dashboard without signing in.
Expected Result	The system prompts the user to sign in to GitHub before displaying any data.

4.7 Requirements Matrix

This section provides a cross-reference that traces components and data structures to the functional requirements in the SRS.

Table 4.32: Requirements Matrix

Req. ID	Requirement Description	Class	Status
FR01	The user shall be able to view detected code smells in the system.	CustomTreeProvider	Developed
FR02	The user shall be able to view various code metrics on the dashboard.	CustomTreeProvider	Developed
FR03	The user shall be able to access the dashboard to monitor their code analysis progress.	CustomTreeProvider	Developed
FR04	The user shall be able to provide feedback on detected code smells or metrics.	CustomTreeProvider	Developed
FR05	The admin shall be able to review user-provided feedback for system improvements.	-	Developed
FR06	The system shall parse the code provided by the user to extract relevant details.	ASTParser	Developed
FR07	The system shall calculate specific code metrics based on the parsed data.	MetricCalculator	Developed
FR08	The system shall send calculated metrics to the integrated ML model for further processing.	ServerMetrics Manager	Developed
FR09	The system shall receive detection results from the ML model after processing the metrics.	SaveModelPredict-ions	Developed
FR10	The system shall return the prediction results to the user in a comprehensible format.	CustomTreeProvider	Developed
FR11	The system shall update the dashboard with the latest detection results and analysis data.	MetricsSaver	Developed

4.8 System Deployment

The CodePure extension is deployed using Event-driven architecture, leveraging AWS services for model inference and data processing. The system consists of multiple interconnected components, including data preprocessing, machine learning (ML) model training and deployment, API management, and a dashboard for visualizing results.

- The ML model is deployed on **AWS SageMaker**, ensuring scalability and efficiency in analyzing code quality.
- **AWS Lambda** is used for serverless execution, handling requests from the **AWS API Gateway** and interfacing with the ML model.
- **AWS SQS (Simple Queue Service)** is implemented for asynchronous communication between different system components.
- The CodePure extension integrates directly with the developer's code, extracting components, validating syntax, and sending structured data for analysis.

4.8.1 Tools

A variety of tools are integrated into the CodePure system to enhance its functionality:

- **AWS Services**: Includes SageMaker for ML model hosting, Lambda for serverless execution, SQS for message queuing, and API Gateway for managing API requests.
- **VS Code Extension API**: Enables CodePure to interact with the developer's code, parse syntax, and provide real-time feedback.

4.8.2 Technologies

The CodePure system is built on a modern technology stack, including:

- **Programming Languages**: TypeScript (for VS Code extension), Python (for ML and backend services).

- **Cloud Infrastructure**: AWS (SageMaker, Lambda, API Gateway, SQS).
- **Machine Learning**: Feature extraction, model selection, and training using Python-based ML libraries.
- **Web / API Development**: FastAPI for backend services and JSON for data communication.

4.9 Summary

This chapter provided an overview of the system deployment, including the frameworks, tools, and technologies used in the CodePure extension. Additionally, it covered aspects such as the test plan, requirements matrix, and architecture design, outlining the key components and workflow design of the system.

Chapter 5

Evaluation

5.1 Experiments

To evaluate the accuracy and robustness of our model's predictions, we conducted extensive testing using the CodePure extension on a diverse set of medium-sized GitHub repositories sourced from our dataset

The results of these experiments were encouraging for certain smell types and revealed areas for further improvement in others. Data Class detection showed consistently high accuracy, exceeding 95% across nearly all tested repositories. This suggests that the features and heuristics used by CodePure are particularly effective in identifying classes that primarily store data without significant behavior. In contrast, the accuracy for detecting God Classes—classes that tend to centralize too much functionality—was slightly lower, falling between 80% and 85%. This decline reflects the more nuanced nature of God Classes, which can vary significantly depending on the context and design patterns used. The model struggled more with Schizophrenic Class detection, achieving only 55% to 60% accuracy. This is likely due to the complex and often subjective nature of identifying classes that exhibit inconsistent responsibilities or behaviors.

To further validate our model, we extended our testing to include two new external datasets Esmaili et al. [2023], Kachanov and Markov [2023] that were not part of the original training or tuning process. These datasets introduced new coding styles and domain contexts, allowing us to assess the generalizability of the model. On these datasets,

the model achieved 78% accuracy in determining whether a code smell was present or not, and 63% accuracy in classifying the specific type of code smell. While these figures indicate solid performance in unfamiliar environments, they also highlight the need for continued refinement, particularly in improving the classification of more ambiguous or borderline cases.

Overall, the experiments demonstrate that CodePure is a promising tool for automated code smell detection, with strong performance in certain areas and clear opportunities for enhancement in others.

5.2 Performance analysis

To evaluate the runtime and memory performance of our extension, we conducted analysis on approximately ten medium-sized source files. We observed that the first run on a new project consistently incurred a higher processing time of around 11.97 seconds, which we attribute to initialization overhead such as loading parsers, building internal caches, and traversing project structures for the first time. Subsequent analyses stabilized significantly, with processing times ranging between 6.75 and 6.85 seconds per file. Memory usage during analysis remained moderate for most cases, typically ranging between 2 MB and 10 MB, with a few anomalous spikes — including a peak of 363 MB — likely caused by transient memory allocation patterns or garbage collection events inherent to the Node.js environment. Overall, the extension demonstrates efficient and predictable performance post-initialization, making it suitable for integration into real-time development workflows.

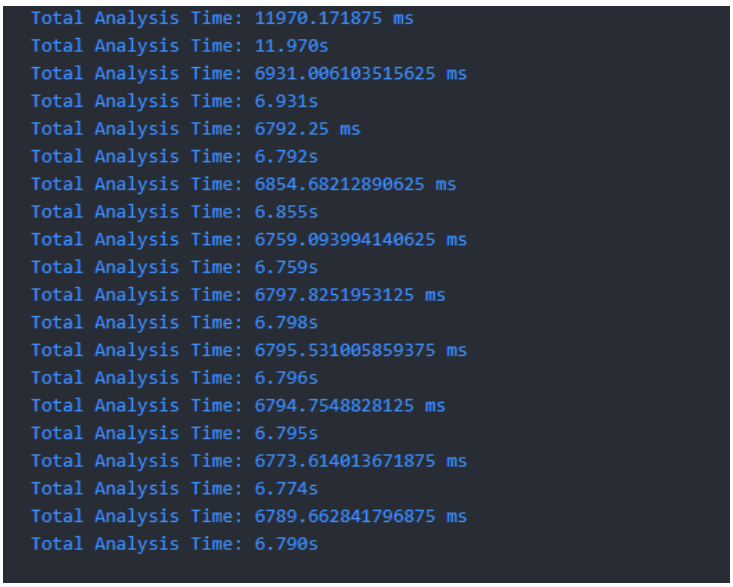


Figure 5.1: Time Taken To Perform A Prediction

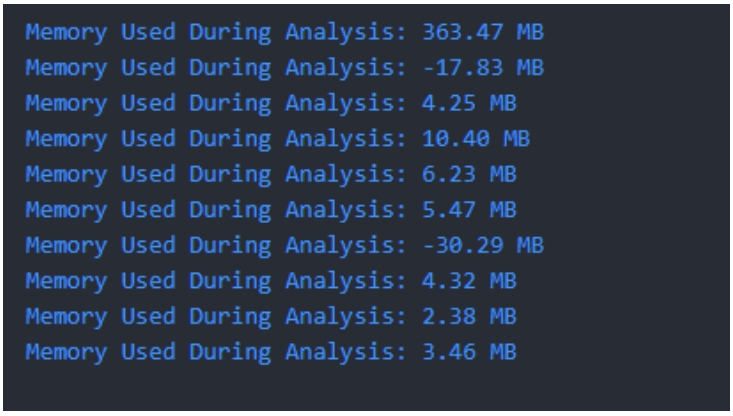


Figure 5.2: Memory Used

5.3 Summary

This chapter evaluated the CodePure extension, focusing on detection accuracy and runtime performance. Testing showed strengths in detecting common code smells like Data Classes but challenges with subjective smells like Schizophrenic Classes. Generalization tests confirmed reliability across different codebases, though improvements are needed. Performance analysis showed higher processing times for initial runs on new projects, stabilizing around 6.75 to 6.85 seconds for subsequent runs. Memory usage was moderate with occasional peaks. These findings highlight CodePure's effectiveness in automated code smell detection and areas for refinement in detection and optimization.

Chapter 6

Conclusion

6.1 Introduction

This project focused on the development of **CodePure**, an automated tool designed to detect and highlight code smells in Java codebases using static analysis and machine learning techniques. By leveraging technologies such as **Tree-sitter** for code extraction, **metrics analysis**, and **ML prediction models**, the project aimed to provide developers with immediate insights into the maintainability and quality of their code.

The major contributions of this work include the extraction of detailed code metrics, accurate code smell detection (specifically for Brain Class, God Class, and Data Class), and the introduction of an **AI Quick Fix** feature that suggests improvements based on detected smells.

Compared to existing tools such as iPlasma and JDeodorant, **CodePure** offers an updated, lightweight, and extensible framework better suited for modern development environments. The use of Tree-sitter enables faster and more robust parsing, while the machine learning-based approach allows the tool to adapt and improve with more data.

The significance of this project lies in its ability to **bridge the gap between traditional metric-based analysis and intelligent, AI-supported code quality enhancement**, offering a practical aid for developers maintaining medium to large-scale codebases.

6.2 Contributions and Reflections

Throughout the project, several key contributions were made:

- **Code Extraction and Metrics Calculation:** A complete pipeline was implemented to extract Java code using Tree-sitter, parse class and method structures, and calculate essential metrics such as LOC, WMC, CBO, and others.
- **Code Smell Detection:** The CodePure extension successfully detects Brain Class, God Class, and Data Class smells based on well-established iPlasma rules and customized thresholds derived from real project data.
- **Highlighting Feature:** An innovative feature was added to visually highlight problematic areas in the code editor, making it easier for developers to quickly locate and address code smells.
- **AI Quick Fix:** A novel feature that leverages AI agent to automatically suggest possible refactorings based on detected smells.
- **Extensibility:** CodePure was designed to allow easy integration of additional code smells or metrics in the future, ensuring long-term usefulness.

Reflections:

The project execution generally followed the planned roadmap; however, some challenges arose. For instance, accurately parsing complex Java constructs with Tree-sitter required deeper customization than initially anticipated. Also, training effective machine learning models for certain smells (like God Class) revealed the difficulty of obtaining balanced, labeled datasets. Despite these hurdles, the project was an enriching experience, strengthening skills in static analysis, machine learning integration, and extension development. Overall, the combination of technical challenges and creative problem-solving resulted in a project that pushed both my engineering and research abilities further.

6.3 Future Directions

While CodePure has achieved its initial goals, there are several opportunities for future work:

- **Support for More Code Smells:** Expanding detection to include smells like Feature Envy, Long Method, Shotgun Surgery.
- **Multi-Language Support:** Extending CodePure to work with other programming languages beyond Java, such as C++ or Python, using Tree-sitter grammars.
- **Improved Machine Learning Models:** Training more sophisticated models (e.g., deep learning-based) on larger, more diverse datasets to improve detection accuracy.
- **Real-time AI Refactoring:** Moving beyond suggestions to fully automated safe refactoring actions, integrated within the development environment.
- **Integration with Popular IDEs:** Offering CodePure as plugins for IntelliJ IDEA, Eclipse, and Visual Studio Code, broadening its accessibility and impact.

6.4 Summary

This chapter summarized the objectives, outcomes, and significance of the CodePure project. It highlighted the major contributions, reflected critically on the project experience, and proposed future enhancements. Overall, CodePure represents a meaningful step towards intelligent, automated support for software maintainability and quality assurance.

Bibliography

Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621. IEEE, 2018.

Ehsan Esmaili, Morteza Zakeri, and Saeed Parsa. Code smells and quality attributes dataset. Figshare, August 2023. URL `https://figshare.com/articles/dataset/Code_smells_and_quality_attributes_dataset/24057336`. Accessed: 2025-05-06.

Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *2011 IEEE fourth international conference on software testing, verification and validation workshops*, pages 450–457. IEEE, 2011.

Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

Thirupathi Guggulothu and Salman Abdul Moiz. Code smell detection using multi-label classification approach. *Software Quality Journal*, 28(3):1063–1086, 2020.

Mouna Hadj-Kacem and Nadia Bouassida. A hybrid approach to detect code smells using deep learning. In *ENASE*, pages 137–146, 2018.

Maen Hammad and Asma Labadi. Automatic detection of bad smells from code changes. *International Review on Computers and Software*, 11(11):1016–1027, 2016.

Vladimir Kachanov and Sergey Markov. Trusted code smells dataset. Zenodo, 2023. URL `https://doi.org/10.5281/zenodo.7612725`. Accessed: 2025-05-06.

Lech Madeyski and Tomasz Lewowski. Detecting code smells using industry-relevant data. *Information and Software Technology*, 155:107112, 2023.

Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5:1–28, 2017.

Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-smell detection as a bilevel problem. *ACM Trans. Softw. Eng. Methodol.*, 24(1), October 2014. ISSN 1049-331X. doi: 10.1145/2675067. URL `https://doi.org/10.1145/2675067`.

José Amancio M Santos, João B Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos Do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes De Mendonça. A systematic review on the code smell effect. *Journal of Systems and Software*, 144:450–477, 2018.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015. doi: 10.1109/ICSE.2015.59.

Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002.

Appendix A

Git Repository

Links:

- `https://github.com/MohammadHishamm/CodePure-extension`
- `https://github.com/OmarHosny18/Jupyter-notebooks`
- `https://github.com/ZAYATY-260/codepure_serverside`

Figure A.1: CodePure Extension

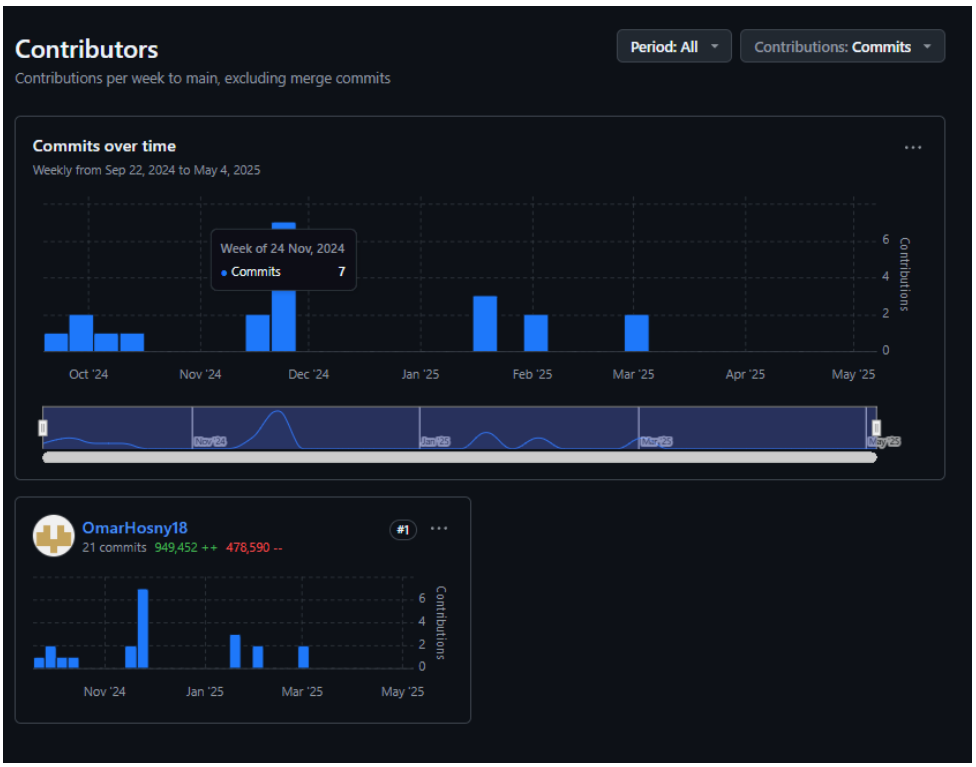


Figure A.2: Jypyter Notebooks

Appendix B

User Manuals

B.1 System Requirements

Before using the CodePure extension, the following requirements must be met on the user's machine:

- Visual Studio Code installed (version 1.60 or later)
- Stable internet connection

B.2 Installation Instructions

The CodePure extension is installed through the Visual Studio Code interface:

1. Open Visual Studio Code.
2. Navigate to the `Extensions` panel (or press `Ctrl+Shift+X`).
3. Search for `CodePure`.
4. Click `Install` on the CodePure extension.

B.3 Using the Extension

Once installed, CodePure runs in the background and automatically analyzes code when supported files are saved.

- Open any Java file.
- Make changes or write new code.
- Save the file (`Ctrl+S`).
- CodePure will automatically start the analysis.

B.4 Understanding the Output

When the analysis is complete, the results are shown in this location:

- **Dashboard Panel:** A sidebar panel will display:
  - The list of detected code smells.
  - Detailed software metrics (e.g., LOC, CC, WMC).
  - Optional feedback button to help improve detection.

B.5 Troubleshooting

- **No results?** Ensure that your file is a supported language (Java), and that it has no syntax errors.
- **No internet connection?** CodePure requires internet access to communicate with the machine learning backend.
- **Extension not working?** Try reloading Visual Studio Code or reinstalling the extension.

B.6 Feedback and Support

Users are encouraged to provide feedback through the dashboard interface. Any suggestions or bug reports can also be submitted via the extension's support page.

Appendix C

User Evaluation Questionnaire

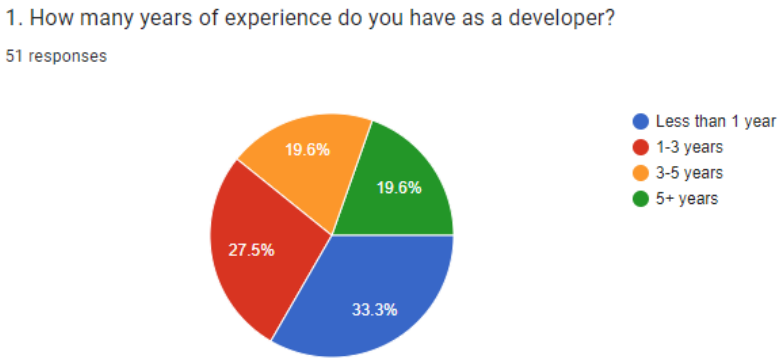


Figure C.1: Results

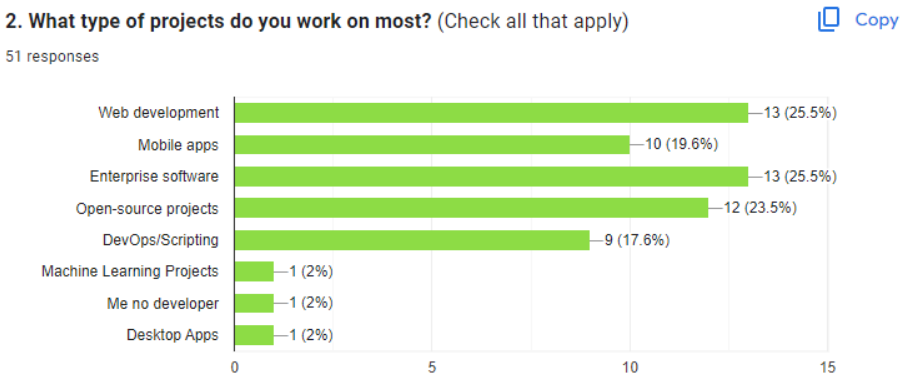


Figure C.2: Results

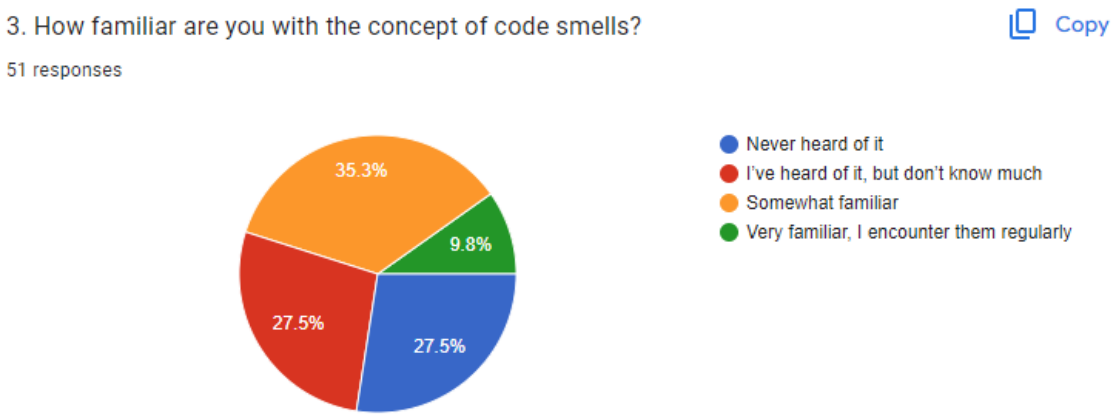


Figure C.3: Results

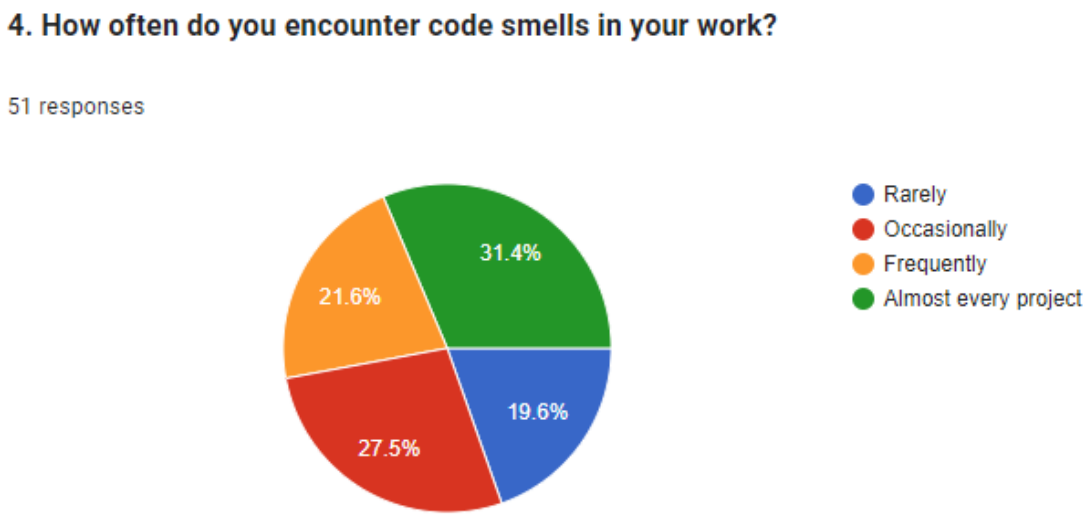


Figure C.4: Results

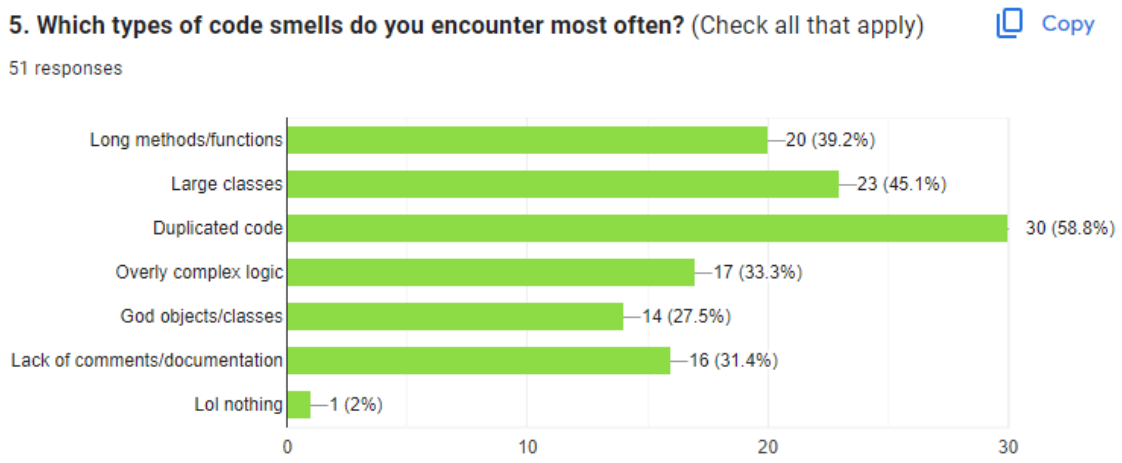


Figure C.5: Results

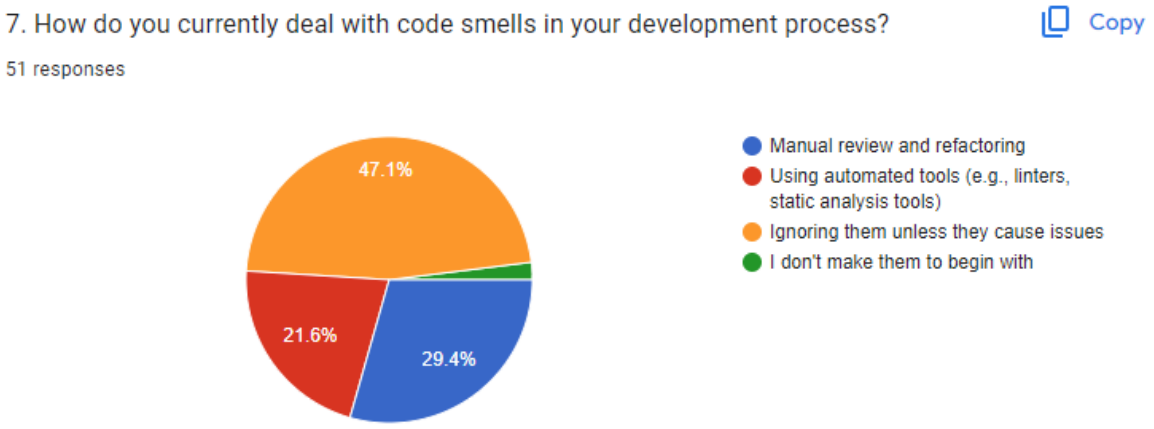


Figure C.6: Results

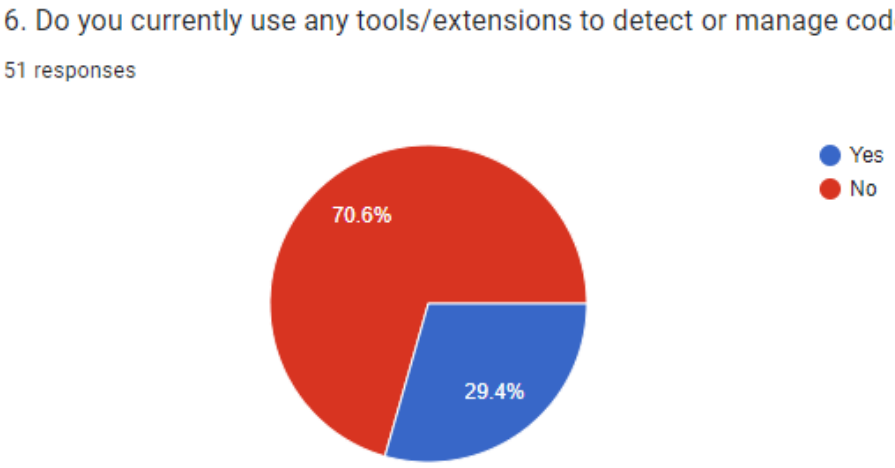


Figure C.7: Results

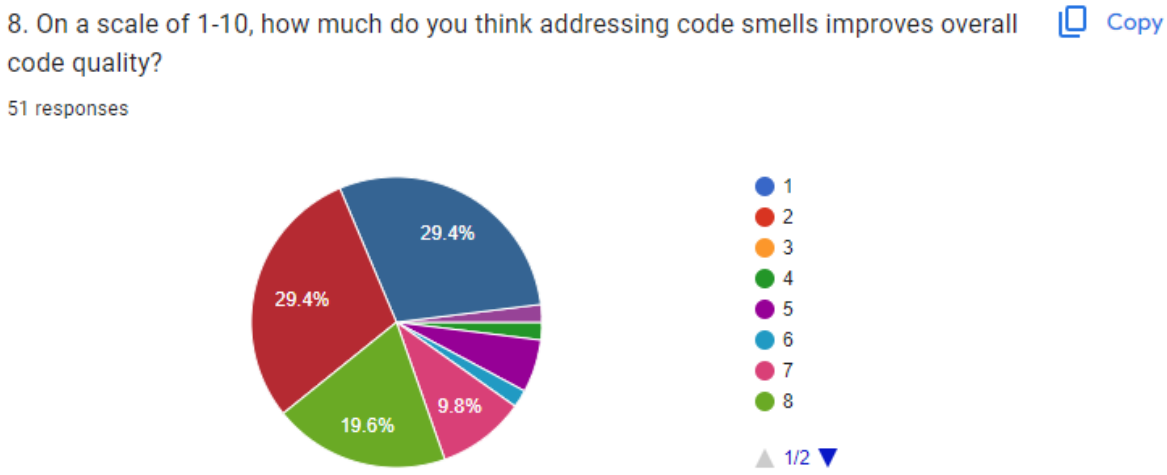


Figure C.8: Results

9. Would you find value in a Visual Studio Code extension that automatically detects and helps you refactor code smells?

Copy

51 responses

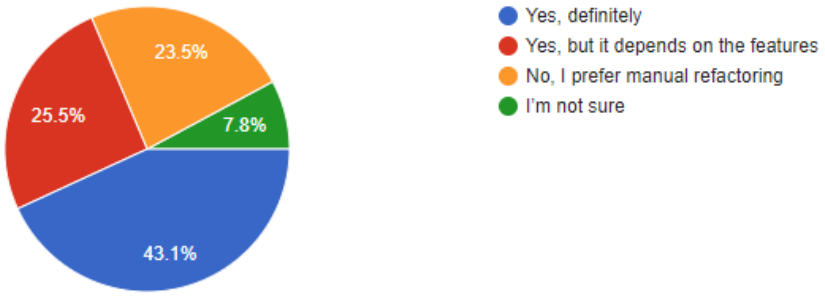


Figure C.9: Results

10. What specific features would you expect from a "Code Pure" extension that helps refactor code smells? (Select all that apply)

Copy

51 responses

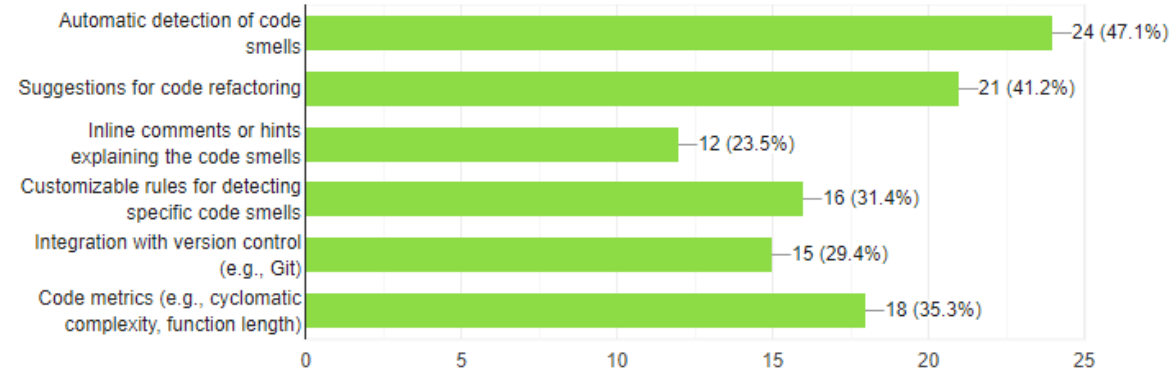


Figure C.10: Results

11. How likely are you to use an extension that focuses on improving code purity by eliminating code smells?

Copy

51 responses

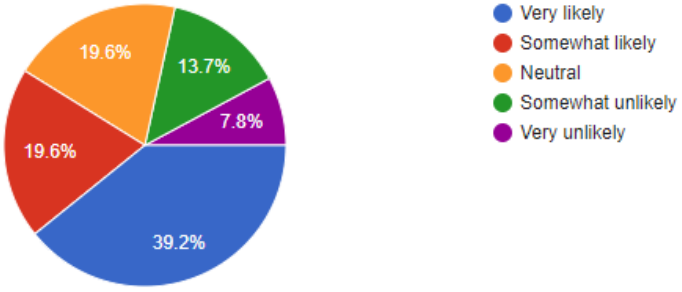


Figure C.11: Results

12. Would you be willing to pay for a premium version of the extension with advanced features?

Copy

51 responses

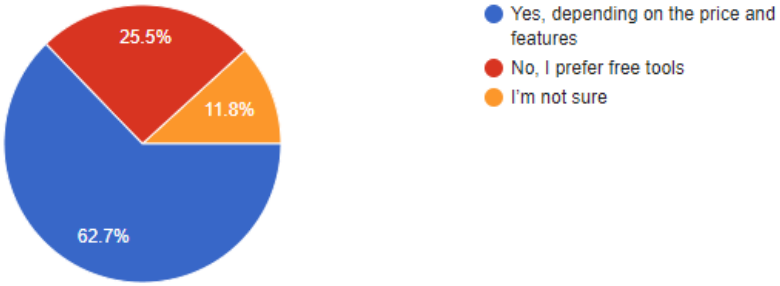


Figure C.12: Results