Improving Structural Code Quality of Java Source Codes Through Automated Detection and Refactoring of Code Smells Leveraging Large Language Models (LLMs)

by

Tahura Alam Daraksha
24241349
Hironmoy Sikder
22101284
Khadija Farhana Esha
24141231
Fardin Kamran
21101023
Ganim Zawhar
21201581

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
Brac University

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Tahura Alam Daraksha

24241349

Hironmoy Sikder

22101284

Khadija Farhana Esha

24141231

Fardin Kamran

21101023

Ganim Zawhar

21201581

Approval

The thesis titled "Smell the Code: Intelligent Automated Detection and Refactoring Leveraging Large Language Models(LLMs)" submitted by

1. Tahura Alam Daraksha (24241349)
2. Hironmoy Sikder (22101284)
3. Khadija Farhana Esha (24141231)
4. Fardin Kamran (21101023)
5. Ganim Zawhar (21201581)

of Spring, 2026 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in June 10, 2026.

Examining Committee:

Supervisor:
(Member)

Md. Aquib Azmain
Senior Lecturer
Department of Computer Science and Engineering
Brac University

Co-Supervisor:
(Member)

Riazul Islam Rifat
Lecturer
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi, PhD

Associate Professor & Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

Structural and design issues in source code, commonly referred to as code smells, can reduce software maintainability, readability, and overall quality. Traditional code analysis tools rely primarily on predefined rules and heuristics for smell detection, which often limits their ability to understand broader code context and provide meaningful refactoring support. Recent advances in Large Language Models (LLMs) have created new opportunities for intelligent code analysis and automated software maintenance. This thesis presents an LLM-based approach for the detection and refactoring of code smells in source code. The proposed framework employs two specialized models: a code smell detection model trained to identify common structural and design issues, and a refactoring model trained on paired before-and-after code examples to generate refactored code and identify the refactoring technique applied. The detection model recognizes several categories of code smells, while the refactoring model generates improved code by applying appropriate refactoring transformations. We are working to propose an automated engine that uses LLM to detect and refactor key code smells in deployed Java projects. Our proposed framework combines static code heuristics with parameter-efficient fine-tuning using for code smell detection. Additionally, a separate LoRA fine-tuned model is trained on before-and-after code pairs to perform refactoring , generating refactored code along with the corresponding refactoring type. We aim to find if the semantic understanding of LLMs are capable enough to find the common smells in deployed Java source codes and refactor to improve the quality of the codes. We will construct a dataset for internal testing and benchmarking. This work illustrates how LLMs can serve as intelligent assistants in software maintenance, enabling automated detection and context-aware refactoring.

Keywords: Code Smell, Technical Debt, Automation, Detection, Refactoring, Maintainability, Readability, Complexity Metrics, LLM, Transformer, Context-awareness, Developer Workload.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In this modern time, software systems are changing at a galloping rate as new features are getting added, defects getting fixed and platforms evolve. Codebases grow and accumulate recurrent structures that hint at deeper design problems known as "Code Smells". Smells are not bugs by themselves, but they raise maintenance costs by making code harder to read, reason about, and modify safely. Although code smell is not a newer concept by any means. The idea of it was first articulated through a systematic study by Fowler et al. in 1999 [2] to refer to pieces of production code that were in poor quality and in need of urgent action to refactor. A few Examples explain Long Method, Dead Code, Too Many Parameters, Duplicate Code, and Commented-Out Code, which are familiarised as bad practices. Some examples of more design smells are God Class, Feature Envy, Data Clumps and large Switch/Type-Checking logic, signal misplaced responsibilities, weak abstraction boundaries and poor cohesion. Static analysis and linters are heavily used to find smells in industrial practice. These tools are deterministic, quick and integratable into the pipelines. These use pre-defined rules and thresholds on metrics (length, complexity, coupling, etc) and syntax trees and control-/data-flow. But fixed rules are problematic in dealing with context and semantics: they easily capture intent, domain idioms, or cross-file relationships. Lots of false positives, poor prioritisation guidance (which rules are most important in a particular context) and little indication of how to resolve problems other than a rule name. Once a smell is detected, a transformation that simplifies or redistributes responsibilities without changing behaviour is a natural companion to detection and is called an automated refactoring. In practice, this is hard. Alharbi [9] classified refactoring approaches into seven categories: Machine Learning (ML) and optimisation-based, graph-based, software metrics-based, rule-based, token-based, text-based and tree-based. Correctness implies behaviour-preserving edits with language-specific constraints; safety implies validation (tests, parsing/AST checks, lints, metrics) to guarantee that the change did not break anything, or simply shift the problems around. There are smells that allow for localised refactorings that can be mechanically checked for correctness (e.g., extract a helper to make a method shorter, remove unreachable code, merge nearly identical methods). Others involve architectural judgment (e.g., moving behavior between classes, creating abstractions), where automated edits are risky and developer guidance is preferable. Over the years researchers have developed a

handful of automated tools which are differentiable according to their capabilities [6], [3], [7].
Recent advances in large language models (LLMs) [4] add a new capability: models can read code and natural language together, reason over larger contexts than other models and produce human-like explanations and patches. However, LLMs are probabilistic and can be inconsistent; they may over-flag, under-specify changes, or propose edits that subtly alter behavior. Any practical system must therefore combine the strengths of static analysis (precision, speed, and determinism) with LLMs' semantic reasoning—and place robust guardrails around edits via automated validation. Industry standard LLMs also lose context awareness as the context gets gradually larger. Programming language characteristics also matter while working on automated refactoring. Python, being an Object-Oriented Language, has a few characteristics such as dynamic typing, multiple programming styles. Idioms complicate purely metric-based rules while offering opportunities for meaningful, localized refactors. Our work here focuses on leveraging LLM engine on code smells refactoring in production code and achieving better refactoring results.

1.2 Rational of the Study or Motivation

The complexity of software production systems is increasing day by day. Developers are required to update their source code constantly [1]. According to Fowler et al., "One of the symptoms of the technical debt are code smells" [2]. As systems evolve, developers unintentionally introduce code smells. While modern technologies such as SonarQube, Checkstyle and PMD already run linters and static analyzers, developers still spend significant time triaging noisy warnings, deciding what to fix, and translating a diagnosis into a safe patch. These technologies are extremely helpful but these are powered by rule-based analysis. Two gaps are persistent:

- **Lack of semantic understanding:** Rule-based detectors struggle with intent, naming, and cross-module context, so borderline cases generate false positives or miss true issues. The results of the long running models back up the lack of LLMs understanding [21]. Additionally, LLMs already detect these smells fairly, but lags behind in refactoring.

- **From flags to fixes:** Most tools stop at reporting; they do not deliver validated, ready-to-apply refactoring or explain trade-offs in the current code context.

These gaps delay maintenance on a day-to-day basis, add to technical-debt carryover, and cause a loss of faith in automated tooling. In contrast, Large Language Models (LLMs) have the ability to understand natural language and programming logic, making them well-suited for context-aware code analysis. These can also generate explanations similar to a human and suggest edits. However, when used alone, they are probabilistic and may over-flag or suggest risky changes [9]. A practical path is a hybrid:

- Static prefilters: Fast and conservative screening and hard safety constraints.

- Context-aware reasoning and patch is elicited by program-generated prompts. without manually creating prompts.

- If any aspect is not up to scratch then fine tuning can not be done.
- Automated validation (tests, AST checks, metrics, and linting) guarantees only safe behavior-preserving edits are applied.

But incorporating this kind of smart analysis in development environments that have to be maintained is difficult. The lightweight and responsive structure is still a challenge. This research is motivated by the need for more intelligent and context-aware code smell detection and refactoring techniques. By leveraging LLM-based understanding of source code and adapting the model through fine-tuning, we aim to improve the accuracy of code smell detection and generate meaningful refactoring suggestions that assist developers in maintaining software quality.

1.3 Problem Statement

Large Language Model based automated refactoration is not at its best yet. LLMs work fairly in detecting but lag behind in refactoring. Additionally, Rule-based tools show quick results but these are not context aware, often create wrong outputs and illogical reasoning. Developers have to back and forth, either prompting in LLM models or updating and fixing constantly. Direct integration into the workload is yet to be seen. LLMs also lack semantic understanding. Furthermore, Production codes are huge lines of code. LLMs also starts to lose context as the context gets bigger gradually.

- **RQ1: Are LLM better at detecting method and class level code smells?**
- **RQ2: How does the quality of LLM-generated refactorings compare to developer-authored reference refactorings across classical OO-design metrics, lexical/semantic similarity, and code-smell counts?**
- **RQ3: To what extent can a QLoRA-fine-tuned 13-billion-parameter code LLM (CodeLlama-13B-Instruct) produce syntactically valid, type-correct refactorings of Java code, and what factors limit its accuracy?**

1.4 Objective

The primary objective of this research is to develop a Large Language Model (LLM)-based system for automated code refactoring of real-world Java projects, aimed at improving code quality by transforming smelly code into cleaner and more maintainable versions. The specific objectives of this work are:

- To identify and collect real-world instances of code smells from open-source Java projects to construct a dataset suitable for refactoring model training.
- To fine-tune a Large Language Model capable of generating refactored code directly from original (before-refactoring) Java source code.

- To enable the model to produce both the appropriate refactoring strategy and the corresponding improved code as part of a unified output.
- To evaluate the quality of generated refactored code by measuring improvements in structural quality and reduction of code smells using an independent code smell detection model.

1.5 Methodology in Brief

Research Approach:
This study follows a machine learning approach, specifically fine-tuned Large Language Models (LLMs), to enhance real-world Java code by detecting code smells and generating code refactored versions of the code. The methodology starts with the preparation of the datasets, which consist of real-world Java code extracted and organised by code smell categories, refactoring techniques and improved code examples. These datasets are then used to fine-tune Large Language Models (LLMs) to learn the characteristics of code smells, as well as the transformation patterns needed for refactoring. The model, after training, is used to process raw Java source code and produce the related refactoring strategy and refactored code. The output is analyzed based on the code smell reduction and maintainability improvement based on an independent code smell analysis model. The overall workflow of the system is as follows: dataset preparation → model training → refactoring generation → evaluation.
Code Smell Identification:
This component applies a Parameter-Efficient Fine-Tuning method to pretrained Large Language Models with code smell annotated datasets. The model is trained to analyze Java source code and classify code segments into predefined categories such as Long Method, God Class, Feature Envy and others that are included in the dataset, to determine whether a code segment contains a code quality issue or not. The fine-tuning process allows the model to learn structural and semantic patterns of various code smells from real-world code examples. The models that can be fine-tuned in this process are called base models, and the candidates include DeepSeek-Coder, CodeT5+, StarCoder2, and Llama 3 (Instruct), all of which have a strong ability to understand code, are easy to fine-tune, and are feasible to compute. The trained model for each input code snippet predicts the most relevant code smell category and an associated confidence score. This learning-based approach is able to detect code quality problems in a consistent and context-aware way, which minimizes the reliance on manually designed rules and enhances the robustness to coding style variations.
Code Refactoring:
This part utilizes a well-tuned Large Language Model to fine-tune the model to produce better versions of the source code leveraging direct transformations learnt from real world examples. The input to the model is an original java program with code smells and its output is a refactored java program and the refactoring strategy learnt from training set. At the time of training, the model is given structured examples that contain the bad code, refactoring method, and good code. This enables the model to learn on how the various code smell patterns are reversed to a cleaner and more maintainable version. However, after training, the model

can produce refactored code that maintains the same functionality as the original code, but enhances the code's readability, maintainability, and design quality. The learnt refactoring strategies are not explicitly requested, but rather adopted "on the fly" in the way the output is generated from the model. By leveraging learning, this approach is able to consistently generate refactored code for various code smell types and lessen reliance on manual prompt engineering or rule-based code transformation techniques. Experimental Design and Dataset: The study employs some famous code smell datasets including DeepLearningSmells, CodeSmell40, github projects etc. The dataset is then preprocessed, normalized and split in a stratified manner. It guarantees a balance in the representation of the types of smells between training, validation and tests.

1.6 Scopes and Challenges

Scope:

- Focused on Java now and extendable to other languages.
- Detection and refactoring are limited to a subset of common smells. Can be expanded to a larger smell catalog.
- The project will emphasize performance, accuracy, and developer usability, evaluated against traditional tools and manual inspection.
- Integration can be performed through an IDE-based interface (VS Code) for interactive feedback, refactoring suggestions, and user evaluation.
- In the future, the implementation of continuous learning from developer feedback can be achieved. The engine could improve over time based on accepted or rejected suggestions.

Challenges: Ensuring consistent LLM responses for varied code structures is a challenge for us. LLMs can produce non-deterministic responses when analyzing and refactoring code, also known as hallucination. Also, there are not enough open-source datasets in this area to properly fine-tune every aspect of the engine. We are only focusing on fine-tuning our detection part; a prompt-based approach will be taken for the refactoring. Developing internal benchmarks that accurately represent real-world developer workflows is necessary but time-consuming. Additionally, measuring the correctness and quality of LLM-generated refactorings is not straightforward. Unlike traditional tools that rely on rule-based metrics, LLM outputs are semantic and context-dependent, demanding a hybrid validation approach using unit tests, AST checks, and metric-based improvements. Furthermore, maintaining smooth communication between the IDE, backend, and model is also a must if we pursue integrating this into an IDE such as VSCode. If we pursue API based implementation, LLMs can handle only limited token input, which may restrict analysis of very large files.

Chapter 2

Literature Review

2.1 Preliminaries

Understanding the key concepts are essential when working with broad topics like this. The concept of Large Language Models and Code Smells introduce vital keywords that are needed to be addressed.

1. **Code Quality Fundamentals**
    - **Code Smell:** An indication of bad coding style that reduces readability and maintainability.
    - **Refactoring:** Rewriting existing code without hurting external behavior.
    - **Technical Debt:** Accumulated cost of code choices that make future maintenance harder.
    - **Static Code Analysis (SCA):** Testing source code without executing to figure out potential issues.
    - **Dynamic Analysis:** Executing final product to analyze runtime behavior and flaws.
2. **Machine Learning and Large Language Model**
    - **Large Language Model (LLM):** A neural network trained on understanding and generating human-like text.
    - **Transformer:** A neural network based on attention mechanisms. Foundation of LLMs.
    - **Fine-tuning:** Curating a pre-trained model to a specific task using smaller datasets.
    - **Hallucination:** Where an LLM model generates irrelevant information with confidence.
    - **Prompt Engineering:** Designing structured inputs for a LLM to generate expected outputs.
    - **Retrieval-Augmented Generation (RAG):** Retrieving external information before generation to enhance model responses.

3. **Source Code Concepts**

   - **Abstract Syntax Tree (AST):** Syntactic organization of source code represented in a hierarchical tree structure.
   - **Intermediate Representation (IR):** Language independent form of code.
   - **Tokenization:** Breaking the source code into meaningful units also known as tokens.

4. **Detection and Evaluation**

   - **Recall:** Model's ability to find all positive values.
   - **Precision:** Model's accuracy of correctly identified positive instances.
   - **F1-score:** Represents balance accuracy by measuring harmonic mean of precision and recall.
   - **Rule-based:** Detection and refactoration logic based on static rules detached of context-awareness.
   - **Data Imbalance:** Where certain classes (eg. specific smells) are under represented.

2.2 Review of Existing Research

The author Hromadiuk, L. [11] tests Llama-3 8B-Instruct for finding Java code smells: Complex Method, Long Parameter List, Data Class, Multifaceted Abstraction, and Feature Envy, plus "None", and compares it with PMD, Checkstyle, and SonarQube. About 20k labeled snippets (MLCQ, DACOS, DACOSX) are used; method fragments are wrapped into valid .java, tool thresholds are tuned, and the LLM sees full class/file context. The study evaluated different prompting strategies for code smells using LLMs, specifically, it compared role-based prompting techniques like zero-shot, few-shot, chain-of-thought (CoT), and role prompts, scored with precision, recall, F1, etc. Among all the prompting types, CoT performs best and achieves strong results. In comparison with traditional static analysis tools, Llama has high recall but more false positives. However, its scope is limited to Java only, leaving cross-language code behind.

Mahbub R. [20] compares simulation and traditional Java software through two studies: (a) how often code smells occur, how long they persist, and whether they relate to bugs, and (b) how effective and risky refactoring is and its impact on maintainability. Using large GitHub datasets [20] (Study a: 155 simulation/327 traditional; Study b: 104/272) and tools such as Designite, RefactoringMiner, PyDriller, and JaSoMe, the analysis applies Mann–Whitney, Kaplan–Meier, Chi-square/Cramer's V, and Nelson–Aalen statistics. Study [20] finds simulation systems have more smells per LOC—notably magic numbers and long statements/parameters—that survive longer, with no significant smell-bug association. The second study shows refactoring removes more smells per LOC in simulation projects, reducing smell survival from approximately 95% to 65% over 5 years. By scope [20], package-level refactors best address implementation/design smells, while method-level refactors work

best for architectural smells; architectural changes carry the highest risk of introducing new smells, and "split conditional"/"move code" can backfire. In practice, teams should focus on magic numbers, long statements, view smells as maintainability debt, prefer class/package-level refactoring, and give careful consideration to method-level changes.

Barroso, S(2024) [10] compares the performance of CodeBERT for the detection of IaC security smells with rule-based GLITCH, addressing data scarcity/imbalance, and enhancing inputs with context and a language-neutral IR. The training corpus [10] spans 196,755 Ansible/Chef/Puppet scripts, with oracle test sets per tech (81/80/80). Labels are created via injection and injection+GLITCH (the strongest), imbalance is reduced by undersampling, inputs include single-line, multi-line (context), and IR, and performance is measured by F1/precision/recall after fine-tuning CodeBERT. Injection+GLITCH outperforms injection alone, and controlling the dominant "None" class markedly boosts F1. Compared with GLITCH, single-line CodeBERT often trails overall but wins on hard-coded secrets; adding context and/or IR delivers large gains, frequently matching or surpassing GLITCH (especially for Ansible/Puppet), while self-training offers no benefit. The key takeaway is that the method [10] uses a Transformer and context (IR) with balanced classes and weak labels (injection and GLITCH) and retains GLITCH as a complementary baseline.

Khleel and Nehéz [12] aim to improve code-smell detection (God Class, Data Class, Feature Envy, Long Method) by using Bi-LSTM and GRU models together with data balancing. The author uses datasets and metrics from 74 open source Java projects and investigates 4 smells, such as God Class, Data Class, Feature Envy, and Long Method. Each models were trained thoroughly and evaluated for accuracy, precision, recall, F1, and AUC. The results were good enough but not ideal.Since balancing, ROS will give the greatest boost, oftentimes nearly full marks, while Tomek Links will assist, but not quite as much. The important thing to take note of is that class imbalance must be addressed first (ROS is a good and simple solution); either model will do, but beware of overfitting when scores approach 1.0.

Another study [14] targets 36 studies of preparing data used for code-smell detection by deep learning and finds that data quality is more important than the proper model. Most work is based on Java and focuses on detecting 'smells' such as Long Method, God/Blob Class, Feature Envy or Data Class, with labels generated by tools, experts or a combination of both.They are based on CNN/RNN/LSTM on code, tokens, ASTs or embeddings and are dealing with class imbalance using SMOTE or undersampling. The major problems include lack of data, lack of generalization between projects, insufficient dataset sharing, noisy/expensive labelling, imbalance and duplicate samples. Some promising strategies are the use of cross-project datasets, pretraining on synthetic data followed by fine-tuning on real data, careful sampling & SMOTE, semi-automatic labelling and a well-defined cleaning/documentation pipeline. In a nutshell, good performance is achieved from having good, non-spurious, well-documented datasets, not from selecting a fancy model.

The author Levya, M [8], designs and experiments with a VS code 'Refactoring Tu-

tor' to assist beginners with finding and fixing common code smells in short guided exercises. The author [8] presents both small, local, and larger, structural enhancements (boolean laundering, magic numbers, improving conditionals, de-nesting, etc.) at levels of difficulty. The tutor [8] was run by about 21 students (362 attempts, 201 successful) and about 50 staff (14 of whom finished all students) with selection based on Pylint/Radon style issues. It is a VS Code extension that has a backend and checks answers with diffs, tolerant diff matching, yanked insertions, regex transformers, triggered hints and a tests-first rule. Completion was overall good, with the most difficult ones being nesting, unpacking the tuples, and renaming of variables, and additional transformers corrected false negatives that were generated due to the auto-formatter. A key takeaway is that diff and transformers can make it a reality to check refactorings in the IDE in a language-agnostic manner; however, further testing and support of formatters/alternate valid answers would be welcome.

Another extensive study [5] compared the code smells ranked by their impact on developers according to their perception, based on the use of machine learning on four smells: God Class, Complex Class, Spaghetti Class and Shotgun Surgery. The authors [5] use DECOR (rules on cohesion, naming, and WMC) to detect God/Complex/Spaghetti and HIST (co-change analysis) for Shotgun, with SPIRIT as the baseline. Data comes from 9 open-source projects (Apache/Eclipse), combining automated commit monitoring with developer surveys to produce 1,332 labeled cases scored from 1 to 5. The model [5] uses size, cohesion, coupling, and complexity metrics, with Random Forest selected for best performance and entropy-based feature ranking. Results [5] show DECOR F1 ranges between 72% and 85%, and HIST F1 is approximately 72% (AUC 61%). Overall, the proposed Random Forest [5] approach is about 20% more accurate than the baseline for predicting perceived criticality, while Spaghetti Class performs similarly to the baseline.

Now let's focus on a few recent benchmarking of current industry standard LLMs. Sadik et al. [23] compares the effectiveness of OpenAI GPT-4.0 and DeepSeek-V3. It sheds light on the strengths and weaknesses of these models and provides a cost-efficient solution for software developers while having multilingual insight. The benchmark was made using the evaluation metrics of precision, recall, and F1-score. The detection was further pinpointed across categories and types. And it finishes off with a cost comparison between ChatGPT-4.0's token-based pricing and DeepSeek-V3's complexity-based pricing. GPT-4.0 showed good precision across most of the smells, where DeepSeek-V3 fell short in terms of accuracy. However, both models struggled in Recall, especially in complex smells such as Change Preventers. Furthermore, for large-scale analysis, DeepSeek-V3 was more affordable despite producing higher false positives. On the other hand, GPT-4.0 was more precise as well as more expensive for larger codebases.

Another study from 2025 [16] examines the potential and challenges of implementing LLMs into code refactoring. Refactoring is a practice of improving code quality without hurting the functionality. The study delves into the surge of interest in automated code refactoring leveraging LLMs. Importantly, author also analyzes the limitations LLM models face, including errors, hallucinations, and integration challenges with IDEs. The authors showed a lack of integration with user-specific needs in mind. Risk of generating syntactically correct but semantically incorrect

refactorings. In addition, LLMs lack an explicit rationale behind refactoring actions. Despite the issues, LLM-generated refactoring still has productivity. The main gap is the lack of a contextual understanding and a validation mechanism is not present here. The authors suggest curating specialized datasets and enhanced LLM prompts to make the refactoring more context-aware.

In the context of established tools, this empirical study [18] investigates the sheer ability of Large Language Models (LLMs) to detect and refactor test smells in test code. The author aimed to achieve automation and improvement of software testing practices. They compared the performance of GPT-4 Turbo, LLaMA 3 70B, and Gemini 1.5 Pro. Junior, S. et al. [18] used two established tools, PyNose (Python) and TsDetect (Java), while covering 15 types of smells. The refactoring process was measured by the smells successfully refactored and the impact on test coverage. The detection accuracy of Gemini is 74.35% for Python and 80.32% for Java. The performance of GPT-4 Turbo was slightly lower than Gemini, with 69.2% for Python and 79.9% for Java. LLaMA achieved the lowest performance, particularly in the case of complex smells (Python: 54.9% and Java: 82.6%). On the other hand, Gemini led in refactoring effectiveness, successfully removing 14.61% of Assertion Roulette in Java, compared to 10.54% by LLaMA and 16.02% by GPT-4. Gemini improved test coverage for Java by up to 1133 lines, while GPT-4 and LLaMA often reduced coverage, with GPT-4 reducing coverage by -574 lines in Java. Python projects showed negative impacts on coverage, with Gemini causing a coverage loss of 65,868 lines in Assertion Roulette, while LLaMA and GPT-4 followed with -64,360 lines and -34,898 lines, respectively. The models showed inconsistent performance across languages. Test coverage degradation was a noteworthy concern, especially with LLaMA. The model showed hallucinated results, which might be because of the lack of open-source curated datasets.

Pandini et al. [22] investigates the feasibility of Large Language Models (LLMs), which are used to identify and refactor architectural smells. Traditional tools are often limited to identifying code-level smells, whereas this study shows the capabilities of LLMs, such as identifying architectural smells, exploring the reasoning, and refactoring them. The authors studied real-world architectural smells such as cyclic dependencies, god components, and unstable dependencies. The study explores using multiple LLMs like Claude 3.5 Sonnet and Qwen to generate refactoring suggestions. The author analyzed 15 circular dependencies(CDs) from various open source Java projects. Each CD was refactored under different prompts using Claude Sonnet and Qwen. The responses were evaluated by master's students and developers. The results show that detailed prompts led to more accurate suggestions for refactoring. During the evaluation process, 78% participants preferred answers from detailed prompts, and 73% preferred retrieval-augmented generation (RAG) based answers. Claude Sonnet showed more consistency while Qwen produced hallucinated or incomplete outputs. The study was limited to a small dataset, and the approach has not been tested in real-world industrial systems. The key limitations were the limited understanding of LLMs of the global architectural context, the sensitivity of prompting, etc. The study does not deeply address the challenges, and there is no explanation of the hybrid approaches combining LLMs with static analysis or architectural metrics.

D´ıaz Arrieta et al. [17] presents a detailed bibliometric study to explore the relationship between code smell and technical debt, two interrelated concepts related to software quality and maintainability. Code smells are generally indicators of poor code quality and design that can trigger technical debt by raising maintenance costs. The author applied bibliometric methods to analyze publication counts, citation networks, co-authorship, and keyword co-occurrence by using data from the Scopus database and visualizing tools like VOSviewer. In the last decade, the number of publications has shown growth, especially in countries like Brazil, Italy, the United States, and Spain. The research is centered on software engineering and maintenance-related journals and conferences with common topics like code smell detection, refactoring, software quality management, technical debt management, etc. However, the research is heavily focused on Java-based open source datasets and limited studies in other languages or cross-language systems. The study also points out several research needs, such as standardized metrics, a better understanding of handling code smells, and technical debt by developers. Future work involves exploring AI-based detection techniques for further enhancing accuracy and automation. Overall, the result shares a clear idea of how research has developed in the past decades on code smell and technical debt, emerging areas like AI/ML-based detection.

Another study by Lin et al [19] introduces a new framework named FlowGen that uses LLM agents to simulate traditional software development process models like Waterfall, Test-Driven Development(TDD), and Scrum. Each agent has a specific role in the development process, such as requirements engineer, architect, developer, tester, or scrum master, and they collaborate through a very structured communication pattern to enhance code quality. FlowGen assigns LLM agents and adapts its behaviour based on various process models. Models like FlowGenWaterfall, FlowGenTDD, and FlowGenScrum. The agents use chain-of-thought reasoning and prompt refinement techniques to improve generated code. The system is powered by GPT-3.5, which serves as the underlying LLM, and the framework is evaluated against baselines like RawGPT, CodeT, and Reflexion on benchmarks such as HumanEval, HumanEval-ET, MBPP, and MBPP-ET. The framework was tested against other models like RawGPT, CodeT, and Reflexion using popular programming benchmarks such as HumanEval, HumanEval-ET, MBPP, and MBPP-ET. Among all the other process models, FlowGenScrum outperforms others by achieving high accuracy. The results show accuracy scores - HumanEval = 75.2%, HumanEval-ET = 65.5%, MBPP 82.5%, MBPP-ET= 56.7%. These results indicate that a 15% improvement occurs over the standard GPT-based approach, like RaeGPT. Overall, the work highlights areas for future research that might help in growth. Areas like Diverse Programming Problems, Complex Agent Interactions, and Human Oversight Integration, etc. When LLMs are made to follow formal software processes and work together like human teams, the code they create gets better. The design and code review stages were very good at getting rid of code smells and making error handling better. This shows how important it is for intelligent agents to operate together in a way that is aware of the process.

Another study [15] focuses on improving Large Language Models (LLMs) such as GPT to handle automated code quality improvement with the help of static code analysis(SCA). Although LLMs excel at generating code and refactoring, they often

fail to notice structural and semantic problems like hidden bugs, inconsistent styles, code smells, etc. The approach combines LLM output with SCA tools to detect code's internal structure and make accurate improvements. At first, it is prompted to detect code smells, violations, and bugs. The study compared the performance of LLM and SCA approaches with only LLMs. Evaluation of open source projects showed that the augmented method works better than standalone LLMs in detecting and correcting subtle issues like accuracy, quality, and structural code smells. Refactoring with SCA information was more consistent and better aligned with developers' standards, and more practical from a developer's perspective. One of the major limitations was that the research only focused on a small set of code smells and quality issues. In this research, more complex problems such as architectural or design level smells were left untouched. It is unclear how well the approach would work on other languages because the experiment was mainly on Python and Java.

Another framework named MUARF was introduced in a work [24] from 2025, which is a multi-agent workflow framework. This is a framework to facilitate automation of code refactoring with multiple LLM agents. The motivation of MUARF is to overcome the limitations of traditional refactoring, which is manual and time-consuming, and single LLMs which may fail to handle complex situations, by assigning each agent with specific roles such as detecting, planning, refactoring, etc. These agents communicate in a structured pipeline that ensures appropriate planning and precise code transformation. Datasets of open source projects were evaluated. The effectiveness of refactorings, functional correctness, and code quality improvement. MUARF demonstrated higher accuracy in performing complex multi-step refactoring. The study mentioned that multi-agent coordination reduced hallucinations and enhanced refactoring. However, it was limited to specific refactoring types like rename, extract method, simplification, etc. The primary focus was on Python and Java, raising concerns about cross-language adaptability and IED was not fully implemented, which are limiting real development workflow.

Wu et al. proposed iSMELL[13], a model that combines LLMs with other toolsets for detection and refactoring. Despite showing potential in code analysis, LLMs face limitations (eg., token restriction, hallucination). LLMs also lag behind handling complex smells. iSMELL proposed a Mixture of Experts (MoE) model that integrates numerous smell detection tools with a focus of improving accuracy and scalability. Based on a gating network, MoE model chooses the most suitable toolset for detecting each smell among 7 tools. Once detected iSMELL leverages LLMs to generate refactoring suggestions. iSMELL showed an exceptional F1 score of 75.17%, outperforming industry leading LLMs by 35.05%. This also showed excellent and outscoring results for Feature Envy and God Class detection. iSMELL reduced code smells, improving WMC ad CBO for God Class and Request Bequest. Although the model suggested valid refactoring for Feature Envy with 83.67% precision, false positives lead to lower evaluation score. Token limitation also restrained the model where it needed multiple iterations for God Class.

2.3 Summary of Key Findings

After an overview of numerous related studies and recent works, a few capabilities and challenges are identified. These shed light to areas where advantages are shown and where work are needed. Here are the key findings,

LLMs are increasingly effective in Code Smell department:
Studies [11],[23],[18],[22],[15],[24] indicate recent models such as LlaMA-3, GPT-4, Gemini, Claude, Qwen and iSmell can detect smells with good performance. These sometimes refactor with competitive performance as well. High false positives can be seen despite having high recall [11]. If balanced datasets and contextual inputs are used, that is where transformer-based models like CodeBERT thrive [10].

LLMs face challenge while refactoring:
LLMs generate syntactically correct but semantically incorrect refactorings [16]. Models that proposes multi-agent approaches (eg., MUARF, FlowGen) are restrained to certain refactor strategies. Although they reduce hallucinations [19][24]. Integration into workflow contexts are limited [16],[24]. LLMs token limitations is also an issue [13].

Data preparation is critical:
Poor generalization still remains a challenge, especially when models focus on single framework. Context-rich datasets improve detection accuracy [12],[14]. Class imbalances have to be handled using techniques such as ROS or SMOT.

Prompting strategies significantly affect LLM performance:
Detailed prompts or RAG improves refactor quality [19]. Chain-of-thought consistently outperform zero-shot prompting [11],[19].

Traditional static rule-based tools are not dead yet:
PMD, SonarQube, Designite, RefactorMiner are effective baselines [15].LLMs integrated with these improve detection and refactoring consistency.

Smell priority and impact differs:
Certain smells have higher technical debt than others and persist longer [20],[17].

Cross-language smells still remains underexplored:
Almost all of the studies focus on Java, Python and GoLang. Cross-language generalization is not yet achieved.Architectural smell detection and refactoring has scope but limited because of dataset [22].

Lastly but most importantly, Limited Dataset:
Maximum works are limited to prompt-engineering. Lack of open-source datasets hinder fine-tuning. Detection do have impactful dataset but refactoring do not have much. A thing that researchers struggle with the most.

Chapter 3

Requirements, Impacts and Constraints

3.1 Final Specifications and Requirements

This chapter includes the final system requirements, expected societal and environmental impact, project-management approach, risk-management strategy, and economic feasibility analysis for the proposed code smell detection and refactoring system. Detection-module information is based on the preliminary report, while the refactoring-module details reflect the most recent CodeLlama-13B QLoRA fine-tuning run on the diversified Java refactoring dataset and the associated held-out evaluation procedure.

3.1.1 Stakeholder Feedback and Requirements Elicitation

The proposed system is designed for three primary stakeholder groups: individual developers working with Java code in IDEs, software-engineering teams that require quality checks in continuous-integration pipelines, and software-engineering educators or learners who use clean-code examples for instruction.
The requirements were developed through three complementary sources. First, regular supervisor consultations determined the project scope. It particularly included the decision to combine static analysis with a large language model, the dataset construction strategy, and the evaluation approach. Second, concerns reported in the software-engineering papers highlighted two persistent limitations of existing code-smell tools, which were false positives from rule-based detection and limited guidance on how to correct a detected smell. Third, software-quality and maintenance standards require any proposed refactoring to preserve the behaviour of the original program.
The resulting requirements are traceable to one or more of the following sources: stakeholder need (S), literature-confirmed limitation (L), or recognised standard and regulatory guidance (R).

Table 3.1: Sources used to derive system requirements

Source type	**Description**	**Contribution to the requirement set**
Stakeholder need (S)	Needs expressed by students, developers, supervisors, or software teams.	Defined usability, offline execution, integration, and deployment expectations.
Literature-confirmed limitation (L)	Limitations commonly reported for rule-based smell-detection and refactoring tools.	Motivated multi-label detection, stronger repair guidance, and identity cases for clean code.
Standards or regulatory guidance (R)	Software-quality, maintenance, privacy, and ethical reporting expectations.	Established quality, maintainability, privacy, and behaviour-preservation expectations.

3.1.2 Functional Requirements

The system is divided into two cooperating modules. The Detection Module identifies code smells from Java code, while the Refactoring Module generates a refactored version of the code and states the refactoring type used. A third layer provides the interface through which users or evaluation scripts interact with the system.

Detection Module - Functional Requirements

Table 3.2: Detection module functional requirements

ID	**Requirement**	**Source**
F-D1	Accept Java source code at either method or class level and return one or more detected code-smell labels.	S, L
F-D2	Support at least seven output categories: Long Method, God Class, Feature Envy, Complex Method, Complex Conditional, Too Many Methods, and NO_SMELL.	S
F-D3	Operate offline once the model adapter has been loaded locally.	S, R
F-D4	Provide a confidence indicator with each predicted label, such as a probability or ranked score.	S
F-D5	Accept snippets up to the 4,096-token training context limit and handle longer snippets through controlled truncation.	L

Refactoring Module - Functional Requirements

Table 3.3: Refactoring module functional requirements

ID	Requirement	Source
F-R1	Accept a Java method or class and produce a refactored version intended to address one or more identified smells.	S, L
F-R2	Return a structured response containing the announced refactoring type and the refactored Java code inside a clearly delimited Java code block.	S
F-R3	Support Refactoring: None when the input code is already structurally acceptable and should remain unchanged.	L
F-R4	Preserve program behaviour by avoiding semantic changes unless such changes are required by the selected refactoring type.	R
F-R5	Run inference on a single consumer GPU with approximately 80 GB of VRAM after quantisation.	S
F-R6	Support retraining through QLoRA on commodity cloud GPUs such as A100 or H100 80 GB devices without full-precision base-model training.	S

Integration and Plugin Functional Requirements

Table 3.4: Integration and plugin functional requirements

ID	Requirement	Source
F-I1	Provide a command-line evaluation interface for batch assessment on the held-out test set.	S
F-I2	Provide an interactive mode for testing individual Java snippets.	S
F-I3	As future work, provide a VS Code extension with a single refactoring command that returns the model response directly in the editor.	S, L

3.1.3 Non-Functional Requirements

The non-functional requirements specify the expected quality attributes of the system. These requirements include performance, reliability, usability, maintainability, portability, privacy, and resource use.

Table 3.5: Non-functional requirements

Category	Requirement	Target
Performance	Detection latency per Java snippet on a consumer GPU.	$\leq$ 1 second for snippets of $\leq$ 1,000 tokens
Performance	Refactoring latency per snippet on H100 or A100 hardware.	$\leq$ 30 seconds for the 95th-percentile input length
Performance	Training throughput on H100 SXM 80 GB.	$\geq$ 5,000 tokens per second
Reliability	Detection accuracy on the held-out test set.	$\geq$ 75% overall
Reliability	Refactoring response parse rate.	$\geq$ 90% of predictions should contain a readable refactoring label and code block
Usability	Output should be understandable without additional processing.	Readable response format
Usability	Interactive mode should handle memory or parsing failures safely.	Recover gracefully and avoid mid-run crashes
Maintainability	Training, preprocessing, inference, and evaluation logic should remain version-controlled.	All major scripts stored under the project repository
Maintainability	Reproducible execution should be supported.	Fixed seed = 42 and deterministic decoding with temperature = 0
Portability	The system should run across the main development environments used in the project.	Windows, WSL2/Linux, and cloud environments
Security and privacy	User source code should remain on the local machine during inference.	No telemetry and no remote inference required
Resource use	A full retraining run should remain within the planned GPU-time budget.	$\leq$ 50 hours wall-clock

3.1.4 Constraints

The final design was influenced through several practical constraints. Model selection, dataset preparation, training configuration, and deployment planning were determined by these constraints.

1. **Hardware:** Primary local development and inference were designed around an RTX 3080 Ti with 12 GB of VRAM. Larger training and evaluation runs used A100 or H100 80 GB cloud GPUs. The use of 4-bit quantisation, QLoRA, gradient checkpointing, and bounded sequence lengths allowed working with this resource limit.

2. **Data:** The refactoring dataset was constructed from mined open-source Java repositories. As a result, the current system is Java-specific and is limited to refactoring types present in repository histories.

3. **Licensing:** The project uses open-source base models, open-source datasets, and permissively licensed software libraries.

3.2 Societal Impact

The proposed system addresses two things that increase cost in software engineering, maintenance effort and developer onboarding. Software maintenance accounts for a large share of a software cost. Code smells further increase that cost by making programs harder to understand, modify, and test. A tool that identifies problematic structures and provides concrete refactoring suggestions can reduce the time needed to understand and improve legacy code.
The system is also relevant to onboarding. New developers often spend considerable time learning unfamiliar project structures. Common structural weaknesses can

be identified by their type label and they can be shown how to typically solve the weakness by type-labelled refactoring suggestions. The proposed design is not intended to replace the elements.The LLM-based refactoring can help teams that already utilize static-analysis tools like SonarQube, PMD, Checkstyle or Designite.

3.3 Environmental Impact

The refactoring module was fine-tuned using QLoRA on a single NVIDIA H100 SXM 80 GB GPU, completing three epochs over roughly 185755 dataset. Because the base model was not trained from scratch, the compute footprint was substantially smaller than full large-model training.

Table 3.6: Training-time compute footprint estimate

Quantity	**Value**	**Basis of estimate**
H100 SXM power rating	700 W	Upper-bound value based on the GPU thermal design power.
Estimated CO_2 equivalent	Approximately 3.6 kg CO_2e	Uses an assumed grid intensity of 0.4 kg CO_2e/kWh.

The detection module was trained on smaller models and required substantially less compute. Its contribution is estimated at roughly 0.5 kg CO_2e. The combined training-phase footprint is therefore approximately 4 kg CO_2e. This estimate is conservative because it uses peak-rated GPU power rather than measured average power draw.
The use of QLoRA is environmentally significant because it avoids training a 13-billion-parameter model from scratch. Instead, the system updates only a small adapter while keeping the original base weights frozen and quantised. This reduces the training footprint by several orders of magnitude compared with full pre-training.

3.4 Ethical Issues

Both modules were trained using public open source Java repositories. The respective licences of these projects permit such usage, but the original authors didn't express specific consent to the incorporation of their code in a machine-learning training set, and the issue of attribution on machine-learning datasets is not quite done in the community. We only published projects under permissive licenses, to minimize this concern, but it can't be removed completely. A more practical problem is that sometimes the code generated by the refactoring model will compile without distinguishing the behaviour of the code. As such, each model output is considered a suggestion for the developer to review before committing, and the system design is deliberately geared towards aiding the developer's work, not supplanting it. This also mitigates the concern that over time such tools can impact the learning of refactoring practice by junior developers as part of the manual work becomes delegated to the model. Finally, the training corpus reflects the ecosystem of all public Java projects, with specific project sizes, coding styles and developer communities being over-represented. This limitation should be kept in mind when interpreting

the model's output, especially for projects that are very different from the typical open-source project.

3.5 Standards

The system has been created on the basis of the practices of software quality, software maintenance, static analysis and ethics reporting. These standards and codes assist with maintainability, refactoring safety and reporting transparency.

3.6 Project Management Plan

3.6.1 Resource Identification

Table 3.7: Resource identification

Category	**Resource**	**Source**
Cloud hardware	RunPod H100 SXM 80 GB for training and evaluation.	Rented
Cloud hardware	RunPod A100 80 GB for auxiliary inference.	Rented
Cloud hardware	Kaggle dual T4 GPUs for conversion tasks.	Free tier
Software	Python, PyTorch, Transformers, TRL, PEFT, Unsloth, bitsandbytes, FlashAttention 2, PMD, Designite, RefactoringMiner, matplotlib, sacrebleu, rouge-score, and javalang.	Open-source
Datasets	453 public Java repositories and the Tufano et al. refactoring corpus.	Public/open-source

3.6.2 Project Schedule

The following schedule is a summary of activities completed and those planned for report development (up to the thesis submission period).

Table 3.8: Project schedule

Phase	**Activities**
Phase 1 - Foundation	Literature review, problem framing, and supervisor approval.
Phase 2 - Detection Dataset	Repository collection, PMD and Designite labelling, AST parsing, balancing, and data split.
Phase 3 - Detection Fine-tuning	Training and evaluation of StarCoder2-3B and LLaMA-3.2-3B-Instruct.
Phase 4 - Refactoring Experiments	Initial CodeLlama experiments and dataset diversification.
Phase 5 - Cloud Full Fine-tune	H100 80 GB training for three epochs.
Phase 6 - Inference and Evaluation	Memory-resilient inference and multi-metric evaluation notebook.
Phase 7 - Refinement	Address generation-stopping issues and merge targeted second-pass extraction.
Phase 8 - GGUF Export	Weight merge, de-quantisation, and GGUF q8_0 conversion.
Phase 9 - Report Writing	Preparation of final thesis chapters.
Phase 10 - Plugin Prototype	VS Code plugin prototype and user trial.

3.6.3 Budget

Table 3.9: Project Budget

Item	**Cost (USD)**
RunPod H100 SXM 80 GB for training	Approximately $45
RunPod A100 80 GB for auxiliary inference	Approximately $30
RunPod H100 SXM for hardened re-inference	Approximately $15
Kaggle GPU for conversion	$0
Local workstation electricity	Approximately $5
Hugging Face Hub bandwidth	$0
Total direct expenditure	Approximately $95

3.6.4 Resource Management

The focus of resource management was to avoid unnecessary loss of intermediate artefacts, cloud budget and time. Validation runs were performed before full training and inference.These were written incrementally to minimize memory and storage demands because they were very large CSV files and model checkpoints.To maintain the working environment and prevent the need for repetition of setup costs, cloud pods were not terminated, but stopped between sessions. For the same reason, long running commands were to be run in persistent terminal sessions, and temporary disconnections were not allowed to interfere with the training. This reduced the risk of operation when running for multiple hours in cloud.

3.7 Risk Management

Table 3.10: Risk register

ID	Risk	Likelihood	Impact	Mitigation
R1	GPU memory exhaustion during fine-tuning could force a partial retrain.	High	High	Used QLoRA, gradient checkpointing, FlashAttention 2, length limits, and validation runs before full training.
R2	GPU memory exhaustion during inference on long prompts could interrupt evaluation.	High	Medium	Implemented cascading fallback, reduced generated length when needed, incremental output writing, and skip-and-continue behaviour.
R3	The model may continue generating after the first valid response, reducing parse quality.	Realised	Medium	Identified during evaluation; addressed through explicit end-of-sequence handling and planned re-evaluation.
R4	Model conversion to GGUF may fail because of quantised-weight or library-version incompatibilities.	Realised	High	Applied runtime patching, layer-by-layer de-quantisation, configuration cleanup, and documented conversion steps.
R5	Class imbalance in the dataset could bias the model toward frequent refactoring types.	High	High	Applied a 15,000-example per-type cap, deterministic sampling, and a small set of identity examples.
R6	Cloud budget could be exceeded if long runs are left unattended.	Medium	High	Set phase-level spending ceilings, monitored runs, and stopped cloud pods when sessions ended.
R7	Partial inference outputs could be lost if the cloud pod terminates.	Medium	Medium	Wrote predictions incrementally and enabled resume behaviour based on completed example identifiers.
R8	Downstream redistribution could violate upstream model licences.	Low	Medium	Planned release documentation will state the parent licences and redistribution constraints clearly.
R9	Similar repositories or examples could appear across training and test splits.	Medium	High	Used stratified splitting and deduplication by before-code and refactoring type; residual leakage risk is acknowledged.
R10	Team-member availability could affect progress near the end of the thesis timeline.	Medium	Medium	Distributed responsibilities across detection, refactoring, plugin planning, evaluation, and report writing.

The project was decided in a risk conservative manner. Only risks that could be mitigated at relatively low cost were taken. The three safeguards below were important.

- Validation runs are mandatory: Test runs were performed before multi-hour training or inference jobs, thus making it less likely to have costly configuration failures.
- Automatic resume behaviour: The evaluation scripts were written to resume from last example that was completed instead of starting again from the beginning.
- Checkpointing: The training script stored checkpoints at every 250 optimisation steps and used only the latest checkpoints to manage storage.

3.8 Economic Analysis

Table 3.11: Cost Breakdown

Cost Category	Sub-category	Unit Cost	Total (USD)
Direct	RunPod H100 SXM training	$3.29/hour	$42.77
Direct	RunPod H100 SXM evaluation	$3.29/hour	$16.45
Direct	RunPod A100 inference during initial run	$1.39/hour	$30.58
Direct	Local electricity	Approximately $0.03/hour	Approximately $5.00
Direct subtotal			Approximately $95
Indirect	Workstation amortisation	$1,500 over 3 years	$250
Indirect subtotal			Approximately $6,000
Total project cost			Approximately $6,100

The main advantage of the system is to allow less effort to be expended by the developer to detect and fix maintainability problems. An optimistic estimate is that a developer will choose the refactoring suggestion 3 times every workday for 220 workdays of the year. Assuming that each suggestion accepted will save 12 minutes of manual refactoring and review time, and a loaded developer is $50 per hour, the annual productivity savings per developer is:

$$3 \text{ suggestions/day} \times 220 \text{ days} \times \frac{12 \text{ minutes}}{60} \times \$50/\text{hour} = \$6,600$$

Based on this estimate, the project will be break even after approximately one active developer-year of use against the approximate $6100 total project cost. If the system can therefore provide useful recommendations, the project cost, even if it is limited to a small engineering team, can thus be justified.

3.8.1 Cost-Benefit Estimation

A simple sensitivity analysis illustrates how adoption level affects project return. These scenarios are intended as planning estimates rather than precise financial projections.

Table 3.12: Cost-benefit estimation scenarios

Scenario	Adoption assumption	Average saving per user-year	Project break-even	First-year return relative to $6,100 cost
Pessimistic	5 active users	$1,000	Approximately 6 user-years	0.8x
Realistic	50 active users	$3,300	Approximately 1.8 user-years	27x
Optimistic	500 active users, such as large student adoption	$5,000	Less than 1 user-year	410x

Even the pessimistic scenario approaches break-even within a year. The realistic and optimistic scenarios provide substantially higher returns because the marginal cost of additional local users is very low once the model and scripts have been produced.

3.8.2 Hidden Economic Value

The system may also provide economic value that is not captured directly by the time-saving calculation. Earlier detection of maintainability problems can reduce downstream defect-fixing effort. Type-labelled refactoring suggestions may also support knowledge transfer by teaching junior developers common refactoring patterns through repeated exposure. Finally, local execution reduces vendor lock-in because teams do not need to depend on a subscription, an API key, or the transmission of source code to an external service.

Chapter 4

Proposed Methodology

4.1 Model Architecture

StarCoder2-3B

StarCoder2-3B is a code-generative large language model designed specifically for understanding and programming tasks. It is based on the transformer decoder architecture, which uses self-attention and feed-forward layers to optimise programming languages by capturing token dependencies in both natural language and source code. Moreover, with approximately 3 billion parameters, it ensures a balance between computational efficiency and contextual performance. This model has been pretrained on massive multilingual codebases with strong coverage of Python, Java, and C-like languages. StarCoder2-3B is useful for detecting structural code smells, such as long methods and duplicate code, by using long context windows that analyse entire functions or classes. This model excels in code completion, summarisation, and refactoring suggestions when it is guided by structured prompts. As it is a medium-sized model, the moderate size allows a faster interface and lower hardware requirements, which makes it suitable for IDE-level integration and automated pipelines. However, in comparison with larger models, it has a smaller parameter count. Due to that, StarCoder2-3B may struggle with deep semantic reasoning in complex and multifile projects. It may also produce refactored code that appears accurate and follows the programming rules without validating and preserving the original logic and mechanism.

CodeLlama-13B-bnb-4bit

CodeLlama-13B-bnb-4bit is a Large language model built by Meta AI, specifically trained to understand and work with programming tasks. It is derived from the LLaMA model family and uses a transformer decoder architecture that processes input tokens sequentially and predicts the next token. The 13B refers to the model size having 13 billion learnable parameters for ensuring a balance between performance and efficiency. The model can understand code structure, program flow, and design patterns in different programming languages. Additionally, the bnb-4bit version uses a technique called 4-bit quantisation. This version compresses the model to use less memory, keeping the reasoning ability most of the time. The model

is good at analysing complex code structures, suggesting refactoring changes, and explaining code in human-readable languages. It can identify higher-level design problems, specifically code smells like God Classes or Feature Envy. The model being quantised makes it practical for fine-tuning using QLoRA or similar techniques. It can run on regular GPUs and support code reasoning across Python, Java, C++, and more. However, even with compression, the model is slower and more resource-intensive compared to similar models. Reducing the precision may slightly affect accuracy. The model may also suggest refactorings that seem accurate and reasonable, but they are not totally correct unless extra checks are done to validate the output.

PMD

PMD is a static code analysis tool that is widely used across multiple programming languages for bug detection, API misuse, code smells, etc. It works by parsing source files into an Abstract systax trees (ASTs). The rule set of PMD is predefined but extensible, which allows developers to customise rules. Instead of running the code, PMD examines the source code and finds the problems. This tool is fast, reliable, and predictable whcih makes it very good at finding clearly defined problems such as dead code, unused variables, complex logic, overly long methods, etc. It is commonly used in large projects. However, PMD only follows rules and does not understand the meaning or intention behind code. Due to that, it may miss deeper design issues or flag codes that are written intentionally in code. Also, it does not provide or suggest any smart refactoring or automatically fix the detected problem.

DesigniteJava

DesigniteJava is a static analysis tool specialised for Java that identifies design patterns, implementation, testability, and architectural smells by analysing source code without running it. Instead of focusing only on identifying code smells, it analyses the overall structure of the system. It uses AST or object-oriented metrics to detect patterns associated with poor design. This tool is capable of detecting complex design smells such as GOD Class, Feature Envy, etc. Reporting detailed findings on smell type and severity, it helps developers prioritise fixes. To improve efficiency in large repositories, it supports multi-commit modes. However, the tool only works for Java, leaving different programming languages behind. It is also unable to analyse code behaviour while it runs. Due to its reliance on fixed thresholds, it is unable to automatically suggest or apply fixes for detected smells.

QLoRA QLoRA is not a model itself; it is a method used to fine-tune LLMs efficiently, also in a resource-friendly way. The work structure of QLoRA is that it compresses the original model into low memory format using 4-bit quantisation. By that, the GPU memory requirement is reduced while keeping the model's original capabilities and knowledge intact. After compression, QLoRA adds small and train-

able components to specific layer models, which are known as low-rank adapters. Through that, only the small adapters are updated during training, and the original model parameters remain frozen. This helps the model to learn task-specific behaviours without changing core structure. QLoRA makes it possible to fine-tune large models using limited hardware resources. This makes QLoRA suitable for domain-specific tasks like detecting code smells, refactoring suggestions, and analysing software design issues, etc. But the effectiveness of QLoRA depends on the quality of the training dataset, as well as the design of the prompt used for fine-tuning. Since it does not modify the original model's core, it cannot address the original model's limitations either.

4.2 Dataset Creation and Preprocessing

4.2.1 Detection Dataset

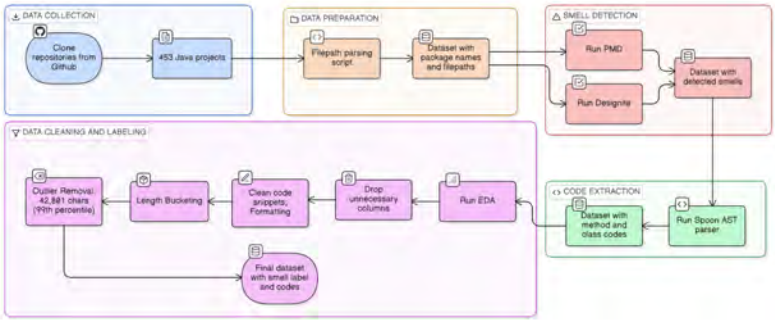


Figure 4.1: Detection Dataset Creation

Our dataset was constructed from 453 open-source Java repositories collected from GitHub. All project directories were traversed to filter ".java" files, and the file paths were recorded. Next, two industry-standard static analysis tools were applied to identify code smells. PMD was used on the Java files to detect common code issues, which parse source files into an AST and match over 400 built-in rules. In parallel, DesigniteJava was used to find higher-level design smells. In our sourced Java code, it detects smell types. Among them, 17 are design-smell types (e.g., Imperative Abstraction, Missing Hierarchy) and 10 are implementation-smell types (e.g., Long Method, Empty Catch Clause, Long Identifier). The outputs of PMD and Designite provide binary flags for each smell type on each code element. We mapped these outputs to our target labels. For example, if PMD flagged the "Long Method" rule on a method, we set its smell label to LongMethod, otherwise No_Smell. If multiple smells could be found, then they were included with a comma separation. In this way, each method or class could be associated with multiple smell labels. We had both method and class-level smells in our dataset.

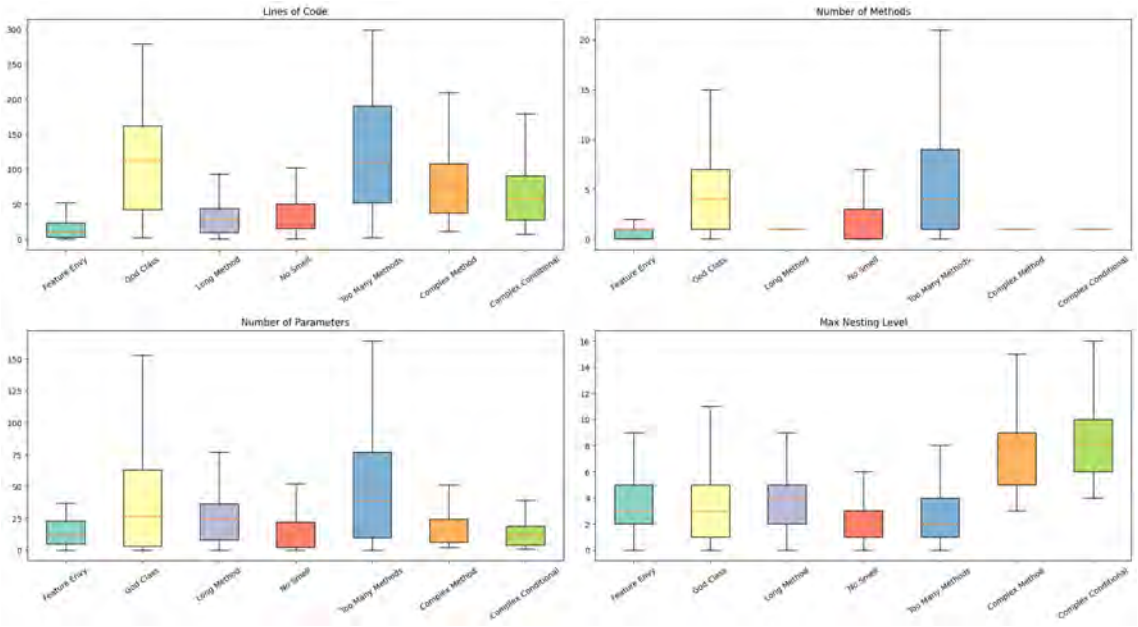


Figure 4.2: Detected Smells

Furthermore, we used an open-source Java metaprogramming library named Spoon for parsing each source file into a complete AST of the code. Spoon also offers an API for analysis. From each method and class, we extracted the full text and preserved the structural information using Spoon. It ensured that we worked with syntactically valid Java code snippets.

Several filtering and normalisation steps were then applied to the raw code examples. To drop extreme cases, we dropped unnecessary columns and removed outliers. As the model can be biased by very short or extremely large examples, we excluded methods shorter than the 1st percentile in length or longer than the 99th percentile. Our model's context window is around 16,384 tokens.
We also shortened the exceeded methods based on our model's context window. Each example was limited to a maximum of 4,000 tokens, which is about 2,000 Java lines to fit memory constraints. Longer methods were cropped from the end. To improve training efficiency, we created length buckets by grouping examples of similar token length (for example, quartiles). Thus, we managed to minimize padding and make better use of GPU computation.

After the cleanup, we examined label distributions. Some code smells were much rarer than others. For example, "God Class" defects occurred in only a few percentage of examples, whereas simple formatting issues were more common. Class imbalance was mitigated through applying data balancing techniques. Minority-label examples were oversampled, and majority-label ones were undersampled. This equalises the frequency of each class. In classification tasks, balancing can be critical due to skewed classes. Models can be misled due to skewed classes.

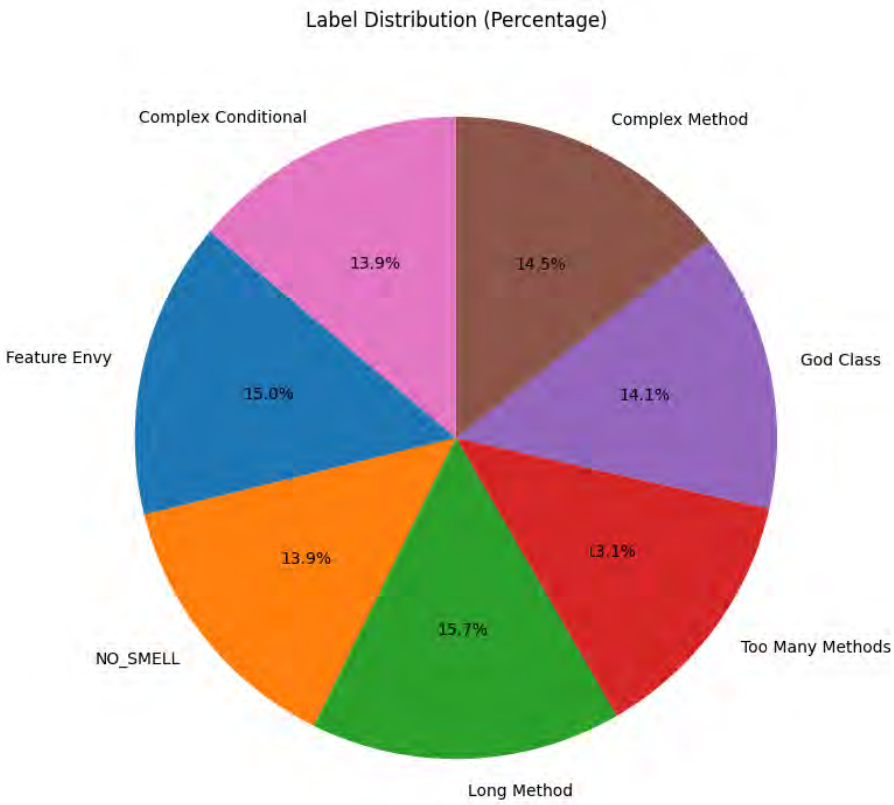


Figure 4.3: Label Distribution

Finally, data were split into training, validation, and test sets in an 80/10/10 ratio. On the smell labels, we used stratified sampling. With that, for each smell, both training and validation sets contain roughly the same percentage of examples. To ensure data quality for the fine-tuning task, each step, such as parsing, detection, filtering, labelling, and balancing, was carefully designed.

4.2.2 Refactoring Dataset

The performance of the proposed refactoring depends heavily on the quality and diversity of the dataset used during training. Hence, a dedicated dataset preparation pipeline was designed to modify the refactored codes that were extracted from Java repositories into a cleaned and structured dataset suitable for instruction-based model fine-tuning.
This section describes the complete dataset preparation process. The pipeline begins with raw Java repositories mined using RefactoringMiner and progresses through extraction, merging, preprocessing, balancing, and final conversion into JSON Lines format.

The preparation process was organised into five main stages:
1. Source data acquisition and structural extraction
2. Targeted extraction of under-represented refactoring types
3. Dataset merging and exploratory analysis
4. Preprocessing

5. Conversion into the final instruction-tuning format

Three main design goals were maintained throughout the pipeline. First, the dataset should represent a wide range of projects rather than being dominated by a small number of large repositories. Second, rare refactoring types that are structurally important should be preserved as much as possible. Third, the final dataset should be reproducible, which means under the same settings repeated executions of the pipeline should produce the same output.

Overview of the Dataset Pipeline

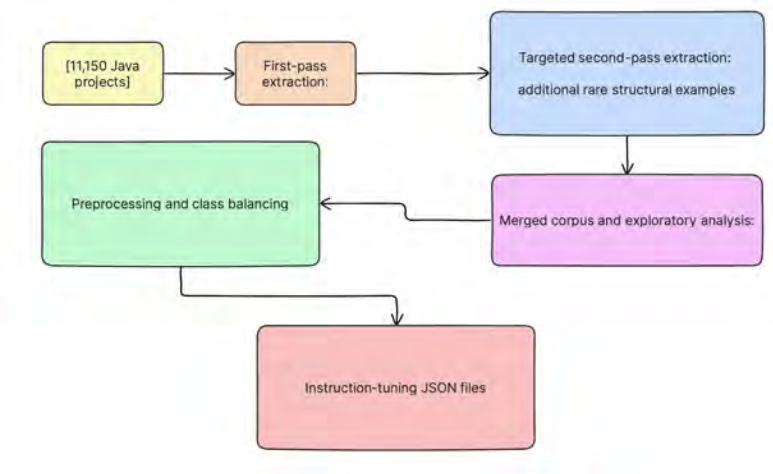


Figure 4.4: Overall dataset preparation pipeline

The overall flow of data from the original repository collection to the final training, validation, and test files.
The first extraction pass was designed to obtain a broad and diverse set of refactoring examples from the repository collection. Then a second targeted pass was introduced that retrieved additional examples of rare structural refactoring types which were under-represented during the first pass. The two outputs were merged and analysed to identify issues such as duplicate records, missing values, class imbalance, and length-related issues. After preprocessing and balancing, the cleaned data was converted into a model-ready JSON format.

Source Data and Provenance

The raw dataset was collected from **11,150 open-source Java repositories** stored locally under the project directory. Each repository had previously been processed using RefactoringMiner, which identifies refactoring operations from software version histories. For each detected refactoring event, RefactoringMiner generated a

set of source files corresponding to the program state before and after the refactoring.

The repository structure followed a consistent layout. Each project contained an output directory, which was further organised by project name, commit hash, and refactoring state. For every commit containing a refactoring, the dataset contained a before-refactoring directory and a corresponding after-refactoring directory. The before-refactoring directory stored the source code before the change, while the after-refactoring directory stored the developer's resulting code after the refactoring.
For each refactoring event, RefactoringMiner produced three files in the before-refactoring directory: the original source code file and two abstract-syntax-tree context files. Only the original source code file was used in this work. The additional context files were excluded during extraction because they were not required for training the refactoring model. The corresponding after-refactoring directory contained the post-refactoring version of the same source file.

Stage 1:First-Pass Diverse Extraction

The first stage was designed to extract a broad and diverse set of refactoring examples from all available repositories. Instead of treating every detected refactoring event as a separate training example, the extractor produced one representative row for each unique combination of commit and source file. This approach ensured the reduction of unnecessary repetition and it also improved the diversity of the resulting dataset.
The extracted row contained the following: source code before refactoring, source code after refactoring, and the selected refactoring type. Four main design decisions were applied during this stage.

Handling repeated refactorings of the same type:

A single file can sometimes contain multiple refactorings of the same type within the same commit. For instance, a developer can commit two methods in a single commit, both of which are extracted from one Java class. Then the before/after code for the events would be the same, and the difference in the dataset would be in metadata, such as the line number.
Repeated occurrences of refactorings of the same kind in the same commit and file were only accounted for by one dataset line, to avoid duplication of this type. This minimised redundancy and kept important refactoring details.

Selecting the most informative refactoring type:

In some cases, more than one type of refactoring was extracted from a single source file in the same commit. The first-pass dataset could only contain one row for each pair of commit-files, and so a selection rule was needed. Hence, the different types of refactoring have been ordered based on their structural significance, and when more than one refactoring type is present, the most informative type has been chosen. The priority order gave preference to more complex structural changes over simpler name changes. For instance, refactorings that involve structure (like Class, Move Method and Extract Method) were found to be more informative than refactorings that simply change a name (like Rename Variable and Rename Method).

Priority categories that were used as part of the first extraction pass were: This

Priority level	Refactoring types
Core structural refactorings	Extract Class, Extract Superclass, Extract Interface, Move Method, Pull Up Method, Push Down Method, Inline Method, Extract Method, Extract Variable, Inline Variable
Secondary structural refactorings	Extract Subclass, Extract And Move Method
Class-level metadata changes	Move And Rename Class, Move Class
Minor structural changes	Parameterize Variable, Replace Variable With Attribute
Rename-based refactorings	Rename Class, Rename Method, Rename Parameter, Rename Variable

priority rule was significant as rename-based refactorings are very common in real software histories. If this weren't the case then the examples in the dataset would have overwhelmingly been name change examples, which are not useful for learning bigger structural changes.

Ensuring project-level diversity:

The size and the extent of the activity on the Open-source projects vary significantly. For larger projects, there can be thousands of refactoring events; for smaller projects, only a few. Without some restriction in the data collection, the resulting data set would be skewed by a few large projects. To reduce this bias, a maximum number of rows was allowed from each project. This limit was initially 50 rows per project in the first pass. Appropriate random but reproducible ordering of commits was assumed within each project. This kept the examples from being skewed toward the older ones or towards alphabetical filename order. This project-level restriction ensured that the data set contained a broader spectrum of coding styles, project structures and refactoring situations.

Incremental writing and memory efficiency:

As each project was processed, the extraction process wrote rows to the output file incrementally. This approach reduced memory usage and made the extraction process more robust. If the process was interrupted, the rows already written to disk were preserved. This was particularly important because the extraction covered more than eleven thousand repositories and produced a large output file.

First-pass extraction results:

The first-pass results are summarised below:

Metric	Value
Total projects examined	11,150
Projects producing at least one row	8,038
Projects producing no usable rows	3,112
Total rows written	200,413

The distribution of the most common refactoring types after the first pass was as follows:

Refactoring type	Share
Move Class	15%
Rename Method	9%
Extract Method	8.7%
Rename Class	7.9%
Rename Variable	6.1%
Rename Parameter	11%
Extract Variable	7%
Extract Class	8%
Move Method	5.4%
Extract Superclass	5.8%
Inline Method	7%
Pull Up Method	5%
Other less frequent types	5.0%

The results show that rename-related refactorings and class-movement refactorings were highly represented. However, several structurally important categories remained comparatively rare. This motivated a second extraction pass focused specifically on under-represented structural refactorings.

Stage 2: Targeted Second-Pass Extraction

Although the first-pass priority rule improved the structural quality of the dataset, it also had one limitation. When multiple refactoring types occurred in the same file and commit, only the highest-priority type was retained. As a result, genuine examples of less frequent refactoring types could be excluded if they appeared together with a higher-priority type.
For example, if a file contained both an **Extract Method** and a **Push Down Method** refactoring in the same commit, the first pass retained only the Extract Method example. For creating a diverse initial dataset,it was acceptable but it reduced the numberof examples available for rare structural categories.
A second extraction pass was introduced to address this. Its purpose was to recover additional examples of under-represented structural refactorings. The second pass differed from the first pass in three important ways.
First, it used a restricted list of target refactoring types. The selected types were those that were structurally meaningful but comparatively under-represented after

the first pass. These included **Extract Class, Move Method, Extract Superclass, Inline Method, Pull Up Method, Inline Variable, Extract Interface,** and **Push Down Method**.
Second, the second pass did not apply the first-pass priority rule. If multiple target refactoring types occurred in the same file and commit, each unique target type was retained as a separate example. This allowed the dataset to preserve multiple valid structural transformations associated with the same code change.
Third, the second pass included a duplicate-checking procedure based on the content of the source code and the refactoring type. If the same pre-refactoring code and refactoring type had already been collected in the first pass, the example was excluded from the second-pass output. This prevented the additional extraction from simply reproducing examples that had already been captured.
The second pass also used a different fixed random seed and allowed a larger per-project contribution limit. This was appropriate because the second pass applied stricter filtering and only retained selected structural refactoring types.
Second-pass extraction results
The targeted extraction substantially increased the number of examples available for rare structural refactorings. The largest gains were observed for refactoring types that frequently appeared together with more common structural operations, especially Move Method, Pull Up Method, Inline Method, and Inline Variable.
The approximate combined totals after the first and second passes were:

Table 4.1: Increase in Dataset Size After the Second Pass

Refactoring Type	**Increase**
Inline Method	4.8x
Extract Superclass	3.9x
Inline Variable	5.9x
Extract Interface	4.4x
Push Down Method	5.4x
Extract Class	3.6x

Overall, the targeted second pass significantly improved the representation of structurally important refactoring categories. This made the dataset more suitable for training a model to perform a broad range of refactoring operations rather than focusing mainly on the most common types.

Stage 3: Merging and Exploratory Analysis

The results of the two extraction passes were merged, and a final corpus was created. The combined data set had about 313,000 rows prior to pre-processing. A duplicate-removal step was performed through a combination of prerefactoring source code and refactoring type. This extra filter was used to eliminate any remaining duplicate examples that could have gone through the previous extraction filters.
An exploratory analysis was then done to get insight into the quality and distribution of the combined data set. The analysis has been carried out for the following attributes: row uniqueness, missing values, distribution of the attribute (refactoring

type), and length of source code.

Row uniqueness
The uniqueness of the first pass dataset was high. Of the 200,413 first-pass rows, 195,722 had a unique value of the source code before it was refactored, meaning that approximately 97.7% of the rows were unique. This has validated the diversification strategy of the project.

Missing values
There were only 44 rows with missing values after the post-refactoring code. These cases were traced back to the incomplete or failure of the refactoring mining process in the case. The number of affected rows was very small compared to the size of the data, so they were deleted in the preprocessing phase.

Source-code length distribution
The combined length of the source code before and after was also analysed. The median combined length was about 6,300 characters, with the 99th percentile at about 19,500 characters. The distribution of examples was skewed to the right with most examples being moderate in length and a few examples having very large class-level refactorings. This analysis was used to determine the length filtering threshold to be applied during preprocessing.

Stage 4: Preprocessing

The preprocessing process consisted of seven steps.
Step 1: Removal of missing values
Rows with missing values in the pre-refactoring code, post-refactoring code, or refactoring type were removed. This eliminated the 44 incomplete rows identified during exploratory analysis and ensured that every remaining example contained the three required fields.
Step 2: Removal of metadata-only refactoring types
Some refactoring types mainly describe changes to project organisation or naming rather than internal code structure. These included:

- Move Class, where a class is moved to another package while its body remains largely unchanged
- Rename Class, where the class name changes but the internal implementation remains the same
- Move And Rename Class, where both changes occur together

These categories were removed because they provide limited training value for a model intended to learn structural code transformation. In many such cases, the source-code body remained almost identical before and after refactoring. Removing these metadata-focused examples eliminated approximately 75,000 rows from the combined corpus.
Step 3: Selection of ten structural refactoring types

The dataset was then restricted to ten structural refactoring types. These types were selected because they involve meaningful changes to code organisation, method structure, inheritance relationships, or variable structure.
The retained refactoring types were:

Selected Structural Refactoring Types
Push Down Method
Pull Up Method
Move Method
Extract Class
Extract Superclass
Inline Method
Extract Method
Extract Variable
Inline Variable
Extract Interface

All other refactoring categories were excluded. This selection ensured that the final dataset focused on transformations that are most relevant to the intended refactoring task. After applying this filter, the corpus contained approximately **182,000 rows**.

Step 4: Removal of low-signal boilerplate code

Typically, Java source files start with package declarations and import statements. These lines are required for compilation, but may not tell much about the actual refactoring operation. Import statements can also be modified automatically as side effects of refactoring, without being a part of the refactoring itself.
Therefore, the pre-refactoring and post-refactoring code had package declarations and import statements removed from them. This minimised the low-signal text in each example and helped to maintain the available token budget for the actual code structure. This removal process was accomplished line by line, and only standalone packages and import statements were affected.
This decreased the total volume of the characters of the data set by around 5%. However, it did not delete any rows, but rather left between 10% and 15% of the files, depending on the file.

Step 5: Removal of examples with no meaningful code change

ASome examples with boilerplate removed were found to be the same before and after refactoring. These cases were generally changes that only involved package/import statements. These examples were not considered as they had no meaningful code change after preprocessing. This was done after removing boilerplate, as some examples are different before boilerplate removal but identical afterwards. These examples would not give a good learning signal to the model.

Step 6: Length-based filtering

Examples were filtered based on the total pre-refactoring and post-refactoring character length. Only rows were kept if they had a total length between 100 characters and 20,000 characters. The low limit eliminated some trivial or incomplete examples that were too small to be of interest. The upper threshold eliminated very large examples, which might deviate from the model's training context limit or require too many training resources. The threshold was chosen to be 20,000 characters, based

on the observed distribution of lengths, and was supposed to exclude the longest outliers while keeping most of the structures. This action eliminated about 7,000 additional rows, reducing the number of rows prior to balancing to about 175,000.

Step 7: Class balancing

The data set remained imbalanced among the 10 types of structural refactoring after filtering. Move Method, for instance, has around 36,000 examples, whereas Push Down Method has around 16,600 examples. This can result in an imbalance that may lead the model to prioritise the most common refactoring categories when trained. Two extreme strategies were considered: Two extreme strategies were used. It is possible to reduce all the classes to the smallest class size and thus produce a fully balanced data set, but this would result in a loss of many useful examples. Or, the data might be kept as is, but then the model would be susceptible to an imbalance. Rather, a compromise solution was chosen. A maximum of 19,100 rows for each refactoring type has been set. Types with >19,100 rows were randomly downsampled with a fixed seed, and types with <19,100 rows were retained. This lessened the amount of imbalance and maintained a significant number of classes compared to an equal class approach. The final preprocessed corpus had 185,755 rows after balancing.

The final distribution was:

Refactoring Type	**Final Rows**
Move Method	19,050
Extract Method	19,025
Extract Class	19,000
Pull Up Method	18,410
Extract Superclass	18,400
Inline Method	18,390
Extract Interface	18,380
Push Down Method	18,370
Inline Variable	18,365
Extract Variable	18,365
Total	**185,755**

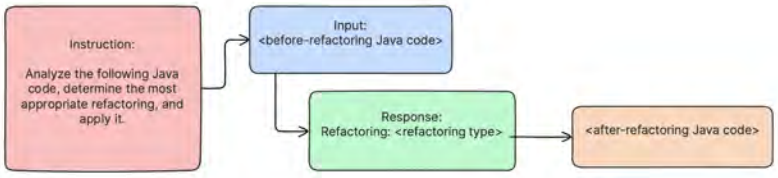
Stage 5: Conversion to Instruction-Tuning Format

The last step involved formatting the cleaned tabular data to JSON Lines for supervised fine-tuning. Each row was converted into a whole instruction-response example. The formatted example consisted of the input code, instructions to the model to perform a refactoring, selected refactoring type, and the refactored code.

Instruction and response format

All the examples of the datasets were formatted in a consistent manner using the same instruction format. The input was a Java program, and the output was of two types: refactoring type and refactored Java program. The structure is simplified as shown below:

The type of the refactored variable was moved into the answer, instead of into



the input instruction. This was an important design decision. It implies that the model can not only execute this code transformation, but also recognise the type of refactoring. At inference, the user should not have to choose a refactoring category manually. Rather, it first guesses the refactoring type, and then provides the refactored code for this type. A pair of code blocks was used for input and output code. This formatting makes it easy to distinguish between instructions written in natural language and the source code, and it also helps with the extraction of the generated code for use later on. For each example, the same instruction was performed on the system level. This consistency enables the model to relate the task format with refactoring behaviour in Java. Examples for no-refactoring: A model trained solely on examples on which refactoring is always needed may learn that every example needs to be refactored. In practice, some code might already be clear enough or structured enough, and the best answer might be "no change". To deal with this, some synthetic identity examples were incorporated into the training set. In these examples, the input and output code were the same and the refactoring type was marked as "None".
The identity examples were extracted from the post-refactoring code of real examples because the code at this stage is a cleaner version of the code after a developer applies the refactoring. Examples of identity were only chosen when the code was not used elsewhere as an example before refactoring, to prevent the presence of conflicting training signals. This allowed the model to not "see" the same code as an example of code that needs to be refactored and an example of code that does not need to be refactored.

Token-length filtering
Each example was then formatted and tokenised with the CodeLlama-13B-Instruct tokeniser. Examples that had more than the configured max sequence length of 4096 tokens were removed. This made sure that the last training examples were not out of the context limit of the model. About 12.6% of the formatted examples were filtered out at this threshold. The average token length for the retained examples was 1,734 tokens, with the 99th percentile being 4,014 tokens. These values suggest that the vast majority of examples will be within the chosen context length.

Stratified dataset splitting
The remaining examples were split into training, validation and test sets with a 90% / 5% / 5% ratio. The split was done by refactoring type (including a synthetic None class). This made sure the proportion of each refactoring category was the same across all three sub-sets. Stratification was especially significant because there were significantly fewer examples of some refactoring categories than others. Rare categories like Push Down Method or Extract Variable would have been underrepresented (or completely missing) from the validation or test set if they had not

been stratified.

4.3 Implementation of Selected Design

4.3.1 Detection

We fine-tuned the StarCoder2–3B,CodeLlama model, a 3-billion-parameter decoder-only transformer designed for code. StarCoder2–3B, CodeLlama uses grouped query attention and supports a very large context window (up to 16,384 tokens) with a sliding-window mechanism of 4,096 tokens. As a language model, the task is formulated as next-token prediction. It is well-suited for sequence outputs like classification tokens. We chose the 3B variant to balance model capacity and resource constraints. It achieves performance comparable to larger models while remaining trainable on a single high-end GPU.

Figure 4.5: Detection Module workflow

To fine-tune efficiently, we employed Low-Rank Adaptation (LoRA), a parameter-efficient method. In LoRA, only small low-rank update matrices are learned on top

of the frozen base model, drastically reducing trainable parameters. Our configuration used rank r = 8 and targeted key, query, value (q, k, v), and feed-forward (gate, up, down) projection matrices within each attention layer. This means each attention block learns a pair of small (rank-8) matrices instead of full weight updates. Empirically, LoRA has been shown to match the performance of full fine-tuning while training far fewer parameters. To further save memory, we loaded the model in 4-bit quantised form (QLoRA). QLoRA combines 4-bit quantisation of the pre-trained weights with LoRA adapters. It was introduced to enable fine-tuning of even 65B-parameter models on a 48GB GPU. In practice, quantising StarCoder2 to 4 bits (with an NF4 data type) reduced the GPU footprint to a few GB, allowing our 3B model to be trained with modest hardware.

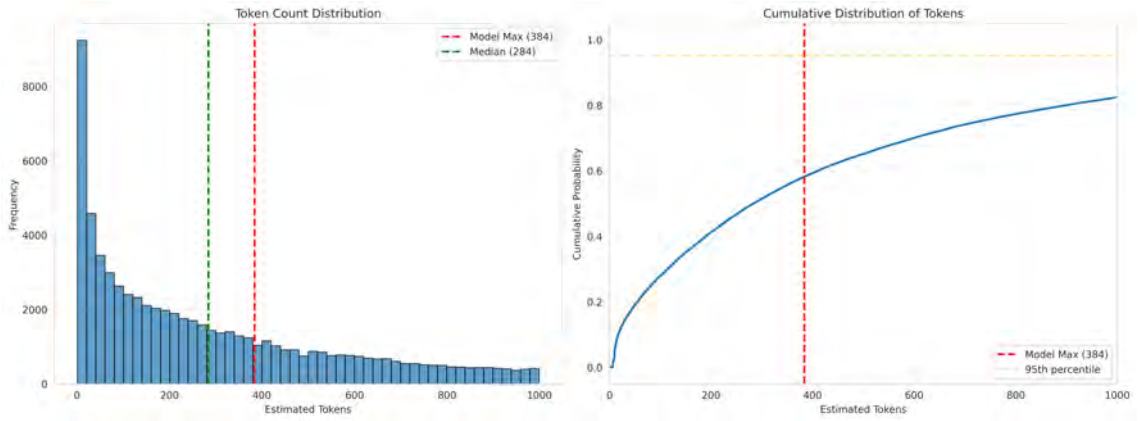


Figure 4.6: Token Distribution

Input formatting and masking were implemented so that the model's loss is computed only on the final "answer" token (the classification label). Using Hugging Face's Transformers-TRL library, we applied the DataCollatorForCompletionOnlyLM. This automatically sets all tokens except the completion to have a label of -100, which is ignored by the loss. We have implemented instruction-tuning for our work, hence we needed a proper prompt construction for Starcoder2. Below is the prompt:

f"""### Task: Classify the code smell in the following code.
Code: code
###Classification:"""

During tokenisation, we include a special token for "Classification:", and the model generates either "True" or "False". Thanks to the data collator, only this last token's prediction contributes to the loss. This encourages the model to focus on the semantic content of the code when deciding the label, rather than modelling the entire prompt along with this setup:

LoRA Rank: 64

LoRA Alpha: 128

LoRA Dropout: 0.05

Batch Size: 8

Number of epochs: 5

Learning Rate: 2e-4

Gradient Accumulation Steps: 4

Gradient accumulation was essential given memory constraints, as it allows a large effective batch without exceeding GPU memory. The optimizer was AdamW with default betas and weight decay. We enabled mixed-precision training (FP16) to accelerate the computation. Mixed precision performs many operations in half-precision (16-bit), which is much faster on modern GPUs. This also reduces memory usage for activations.

However, we enabled gradient check-pointing to further control memory. It saves only a subset of activations and recomputes the rest during backpropagation. At the expense of a 20% slowdown, check-pointing roughly halves the memory cost for storing intermediate states. Model checkpoints were saved when validation loss improved. Finally, we bucketed inputs by length and padded within each bucket to the maximum length in that bucket. This dynamic batching ensures that most of the model's computation is on non-padding tokens, boosting throughput. At evaluation, each project's examples are batched similarly for efficiency, and we compute classification accuracy and F1 per smell label.

4.3.2 Refactoring

Training the refactoring module was done by preparing a pre-trained code-generation model to specialize in Java refactoring. Instead of training a new model from scratch, this work used parameter-efficient supervised fine-tuning. This approach preserves the general programming knowledge present in the base model while teaching it the refactoring patterns required by the proposed module.
This section describes the methodology used to train the module. This includes the selection of the base model, the adapter-based training strategy, the prompt and loss configuration, the software and hardware environment, the main training hyper-parameters, and the safeguards used to make the training process reproducible and resilient to interruption. The empirical evaluation of the resulting model is presented in Chapter 5.

Overview of the Training Approach

The training task was formulated as supervised instruction fine-tuning. Each training example contains an instruction, the original Java code, and the expected answer. The expected answer begins by identifying the refactoring type and then provides the corresponding refactored code. During inference, the model is therefore expected to both choose an appropriate refactoring category and generate code consistent with that decision.
The examples were stored in JSON Lines format and followed the Llama-2 instruction style used by the CodeLlama-Instruct family. The model was shown the complete instruction-response text during training, but the training loss was applied only to the response portion. This ensured that the model learned to produce the refactoring answer rather than simply reproducing the repeated instruction text.

The resulting methodology combines the modelling capacity of a 13-billion-parameter

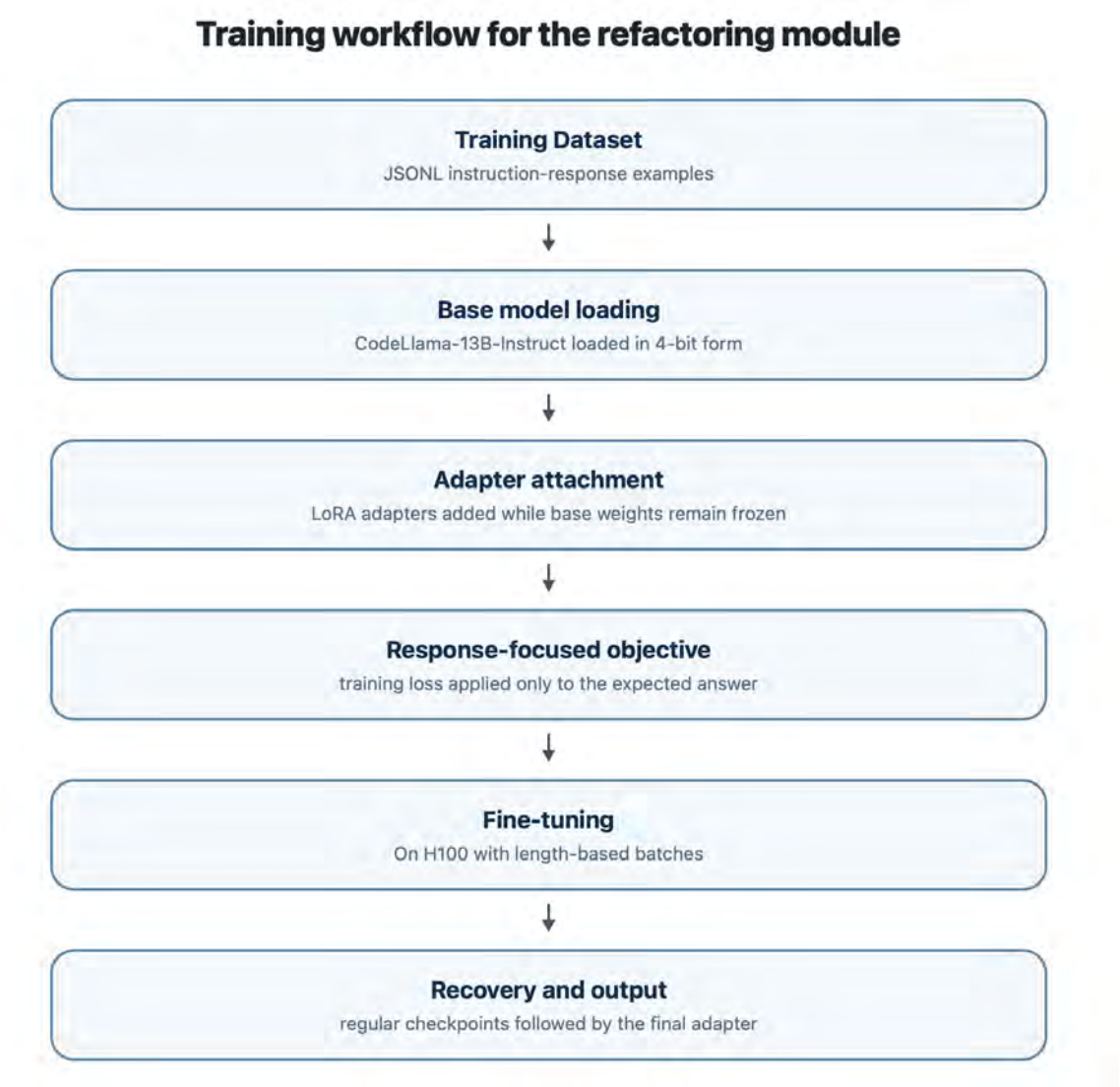


Figure 4.7: Overview of the refactoring module training workflow.

code model with several memory-saving and reliability-oriented techniques. This made the full fine-tuning run feasible on a single cloud GPU while preserving a clear path for reproducible experimentation.

Base Model Selection

The base model was selected by considering three practical requirements: the model needed sufficient capacity to learn structural Java refactorings, it had to remain feasible to fine-tune under the available hardware budget, and it needed to be suitable for later use in an interactive plugin environment. Three candidate models were considered during this stage.

CodeLlama-13B-Instruct was selected as the most appropriate balance between modelling capacity and practical training cost. It provides greater capacity than the

Table 4.2: Candidate base models considered for the refactoring module

Model	Parameters	Approximate 4-bit Memory	Assessment	Decision
CodeLlama-7B-Instruct	7 billion	About 5 GB	Suitable for early experiments, but with limited capacity for more complex structural transformations.	Not selected for final training
CodeLlama-13B-Instruct	13 billion	About 6.5 GB	Stronger model capacity while still practical under 4-bit loading and adapter-based training.	Selected
StarCoder2-15B	15 billion	About 8 GB	Potentially strong, but early experiments showed instability in the selected quantized training pipeline.	Not selected

7-billion-parameter model while remaining manageable when loaded in a compressed 4-bit representation. The larger StarCoder2-15B option was not used because it introduced instability during early tests with the selected quantized training pipeline. The selected model was loaded through the Unsloth pre-quantized distribution unsloth/codellama-13b-bnb-4bit. In this format, the model weights are already stored in NormalFloat 4-bit representation. This avoids an additional conversion step at the start of each training run and reduces the memory required to load the base model. The instruction-tuned form of CodeLlama was important because the training data uses an instruction-response format. Since the base model already understands this general interaction style, the fine-tuning process can focus more directly on the refactoring task and less on learning the prompt format itself.

Parameter-Efficient Fine-Tuning Using QLoRA

Updating all parameters of a model with 13 billion parameters would need a lot of GPU memory. This would take about 26 GB of half-precision model weights, plus another 26 GB of a similar size for the optimiser state. Full fine-tuning would be too costly in terms of local hardware and also inefficient on the cloud GPU used in this work, after considering the activations and gradients. Therefore, the study opted for Quantised Low-Rank Adaptation (QLoRA). QLoRA freezes and stores the original model weights in a compressed 4-bit format. Then, a smaller set of trainable adapter parameters is added to selected model layers. During training, only these adapters are updated. This significantly decreases the amount of memory used, while still enabling the model to learn task-specific behaviour.

Adapter Configuration

The LoRA adapters were inserted into the main attention. This broader placement was chosen so that the adapter could influence both how the model attends to the input code and how it internally represents code transformations. The adapter configuration is summarised in the Table.

With rank 16, the adapters contain approximately 62 million trainable parameters. This is small compared with the 13 billion parameters of the base model, yet

Table 4.3: LoRA adapter configuration used in the final training run

Configuration Item	**Value Used**	**Rationale**
Adapter rank	16	Provides a practical balance between adaptation capacity and memory cost for a 13-billion-parameter model.
Scaling factor	32	Keeps the adapter update magnitude proportional to the selected rank.
Adapter dropout	0.0	No dropout was applied because the dataset is sufficiently large and other regularisation is provided through the learning-rate schedule.
Target layers	q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj	Adapters were applied to both attention and feed-forward projections to support structural code-generation behavior.
Adapter initialization seed	42	Ensures that repeated runs begin from the same adapter initialization.
Gradient checkpointing	Unsloth implementation	Reduces activation memory by recomputing selected intermediate values during the backward pass.

it provides enough capacity to learn the expected refactoring labels and code transformations.

Prompt Template and Response-Focused Training

Each training row was formatted as one complete instruction-response example. The input portion contains the system instruction and the original Java code. The response portion begins with the predicted refactoring type and then provides the refactored code. A simplified representation of the format is shown below.

```
[INST] <<SYS>>
You are an expert Java refactoring assistant.
<</SYS>>

<original Java code>

[/INST]

Refactoring: <refactoring type>

<refactored Java code>
```

The training objective was applied only to the response portion of each example. This design is important because the instruction and input code are already provided to the model; the model should not spend training capacity learning to reproduce them. Instead, the learning signal is concentrated on the expected answer: the refactoring label and the transformed Java code.

Memory and Throughput Optimisation

Managing GPU memory is crucial in order to train a 13-billion-parameter model with 4,096-token examples. Several optimization methods were combined so that the run could complete on a single H100 GPU and could also be tested locally with smaller settings. The most important optimisation was the combination of QLoRA and FlashAttention 2. QLoRA reduced the model and optimizer memory requirements, while FlashAttention 2 reduced the memory needed by the attention computation for long training examples. Gradient checkpointing further reduced activation memory by recomputing selected values during the backward pass.

Hardware and Run Environment

The full training was performed on a single NVIDIA H100 SXM GPU. The GPU was rented through RunPod and had 80 GB of memory. The cloud environment used the RunPod PyTorch template. It was based on Ubuntu 24.04, PyTorch 2.8.0, and CUDA 12.8. The pod also provided 176 GB of system memory and 31 virtual CPU cores, sufficient for data loading and preprocessing during training.

Hyper-parameter Configuration

Table 4.4: Hyper-parameter Configuration Used in the Final Training Run

Hyper-parameter	**Value**	**Purpose**
Maximum sequence length	4,096 tokens	Matches the dataset length cap and supports most formatted examples.
Training batch size per GPU	16 examples	Largest batch that comfortably fit on the 80 GB H100 with 4,096-token examples.
Gradient accumulation steps	1	No accumulation was needed because the target batch size fit directly in memory.
Number of epochs	3	Allows the model to see the full training set three times without an excessively long schedule.
Peak learning rate	2e-4	Appropriate for adapter training, where only a small set of parameters is updated.
Learning-rate schedule	Cosine decay	Increases smoothly during warm-up and gradually reduces updates near the end of training.
Warm-up ratio	0.03	Uses the first 3 percent of steps to stabilise early optimisation.
Weight decay	0.01	Provides light regularisation for the adapter parameters.
Optimizer	8-bit AdamW	Reduces optimizer memory while preserving stable adaptation.
Training precision	BF16	Uses the H100 native mixed-precision format.
Gradient checkpointing mode	Non-reentrant implementation	Improves compatibility with the adapter-wrapped model.
Length grouping	Enabled	Groups examples of similar length to reduce unnecessary padding.
Example packing	Disabled	Keeps each example separate so the response mask aligns correctly.
Dataset text field	text	Identifies the JSONL field containing the complete formatted example.
Data loader workers	8	Uses available CPU resources for data preparation.
Pinned memory	Enabled	Improves host-to-GPU transfer efficiency.
Persistent workers	Enabled	Avoids recreating data loader workers between epochs.
Prefetch factor	4	Keeps several batches prepared ahead of the GPU.
Random seed	42	Supports reproducible training behavior.

The supervised fine-tuning configuration interface of TRL was used to configure the fine-tuning run. Selected values were chosen for a balance between training speed, GPU memory utilisation, and stability. The effective update batch had 16 examples, each with a length of 4,096 tokens. The run had 505 warm-up steps,

which decreased smoothly to zero, whereas the learning rate increased during the first 3 percent of the run.

Length-Based Batching

In a standard batch, all examples must be padded to the length of the longest example in that batch. If a very short example is placed with a very long example, much of the computation is wasted on padding rather than useful code tokens.
To reduce this waste, length-based batching was enabled. This arranges examples so that examples of similar token length are more likely to appear in the same batch. As a result, fewer padding tokens are added, GPU computation is used more efficiently, and training step times become more predictable across the run.
This design was especially important because the dataset included examples ranging from short method-level transformations to long class-level refactorings.

Training Run Behaviour

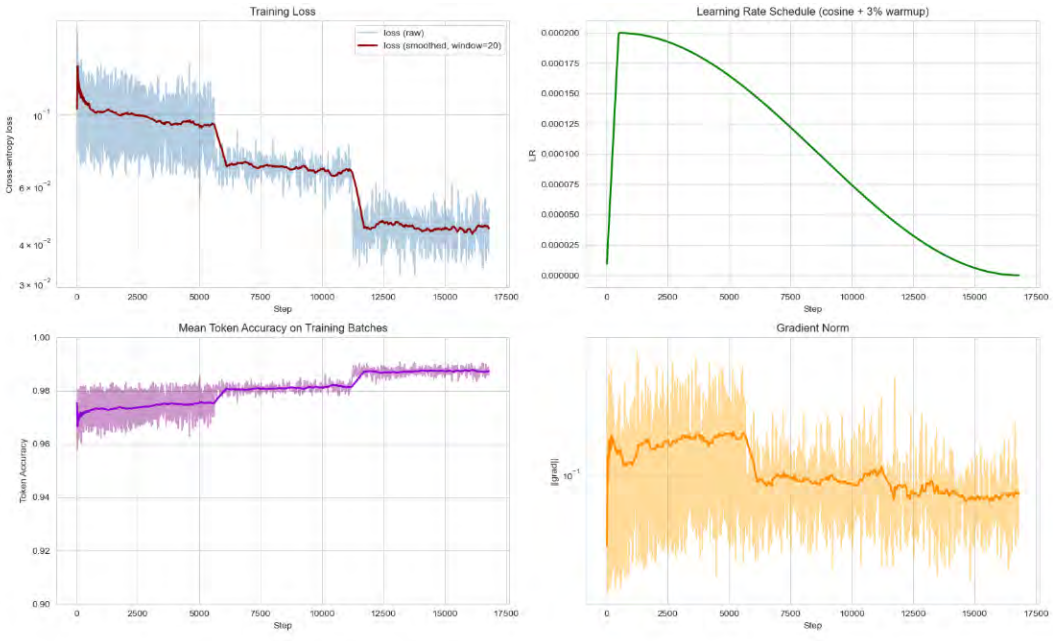


Figure 4.8: Training Curves

The training log contained 672 recorded entries, logged every 25 optimisation steps. The loss curve decreased across the three epochs, indicating that the model successfully adapted to the supervised refactoring objective. The average training loss decreased from 0.098 in the first epoch to 0.068 in the second epoch and 0.045 in the third epoch. At the same time, mean token-level accuracy increased from 97.4% to 98.7%, showing that the model became increasingly accurate at predicting the response tokens used during supervised fine-tuning.
The gradient norm also remained controlled throughout training. The mean gradient norm decreased from 0.143 in the first epoch to 0.080 in the final epoch. This pattern suggests that the optimisation process was stable and that no major gradient explosion or numerical instability occurred. The learning-rate schedule also

executed as expected, following the planned warm-up and cosine decay configuration.

Regime Shifts:

The training loss, token accuracy, and gradient norm display sharp, step-like improvements at Step 5,500 and Step 11,300.

Data-Driven Jumps:

Because the top-right plot shows a perfectly smooth Cosine Learning Rate Schedule (with a 3% linear warmup peaking at 2e4), these sudden performance jumps are not caused by the optimizer. Rather, they denote clear breaks in the training data curriculum, like switching mixtures of datasets or switching batch sizes of sequences. Loss Accuracy: Cross-entropy loss decreases from 0.12 to a minimum of 0.045, and token accuracy increases from 97% to a maximum of 98.8%. Gradient Health: The Gradient Norm (grad) stays well-bounded and variance gets narrower over time, without any catastrophic spikes. This guarantees that the QLoRA adapters converged without optimisation anomalies.

Chapter 5

Result Analysis

This chapter evaluates the proposed refactoring solution using the results produced in the evaluation notebook and the developer-versus-LLM design metric comparison document. The purpose of the evaluation is to assess alternative design solutions, determine the most appropriate approach, and identify the final adjustments required before completing the refactored design. The evaluation focuses on training behavior, code-generation quality, compilation success, refactoring-type accuracy, design metrics, and code-smell reduction.

5.1 Performance Evaluation

5.1.1 CODE SMELL IDENTIFICATION

The model was able to perform classification with a solid classification with an overall accuracy of 75.20% for starcoder2 and 79.76% for Llama3.2.The overall weighted F1 score was 74.90, meaning that the model was able to achieve a somewhat better than average score on all the classes, this along with the Cohen's Kappa score of 0.687 and Matthews correlation co-efficient of 0.688 indicates that there was a significant correlation with the model's prediction along with the test sample.Thus, we can conclude that the model performed classification far better than random guessing.

Table 5.1: Code Smell Classification Performance Comparison

Code Smell Type	**StarCoder2-3B**			**LLaMA-3.2-3B-Instruct**		
	Precision	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Feature Envy	0.8131	0.8177	0.8154	0.830	0.852	0.841
God Class	0.5874	0.5658	0.5764	0.642	0.733	0.684
Long Method	0.7946	0.8797	0.8350	0.790	0.840	0.814
Complex Conditional	0.7820	0.7543	0.7679	0.710	0.730	0.720
Complex Method	0.8064	0.7921	0.7992	0.792	0.780	0.786
Too Many Methods	0.6071	0.5449	0.5743	0.605	0.590	0.597
NO_SMELL	0.8452	0.8413	0.8433	0.850	0.890	0.870

Three categories of code smell showed good performance:

LongMethod
The models achieved the best recall rate and F1-score, and therefore, it is able to

detect methods with excessive length very well. The high accuracy implies that the number of false positives of this type is relatively low.

NO_SMELL
Clean code samples were correctly detected with a F1-score of 84.33 which is a balance and recall. This shows the credible ability of the model in separating between code with and without smells.

FeatureEnvy
This type obtained high performance based on F1-score of 81.54 percent and balanced precision and recall, which means that the model can capture the attributes of methods that overuse features of external classes.

Two types were more challenging to classify:

GodClass
This was the lowest performing category with F1-score of 57.64. The model had difficulties both in precision and recall, which shows that they are not able to differentiate between the classes of god and the other smells. The highest error rate of 43.42 was recorded under all of the categories.

TooManyMethods
This type had the worst performance with the F1-score of 57.43% and the lowest recall. The high rate of error at 45.51% reveals that there is a wide misunderstanding with the other types of code smell especially GodClass.

5.1.2 Refactoring

The first part of the evaluation measures the performance of the fine-tuned refactoring model. The experiment used a CodeLlama 13B QLoRA fine-tuned model and examined the training curves, epoch-level learning behavior, generated-code similarity, compilation success, and performance across different refactoring categories. Since validation evaluation was disabled during training, the training curves are interpreted as indicators of learning stability rather than complete generalization evidence.

Table 5.2: Training Summary by Epoch

Epoch	Logs	Loss Start	Loss End	Mean Loss	Min. Loss	Mean Grad. Norm	Mean Token Acc.
0	224	0.104	0.113	0.098	0.055	0.143	0.974
1	224	0.070	0.067	0.068	0.051	0.095	0.981
2	224	0.054	0.051	0.045	0.032	0.080	0.987

The training summary shows a consistent improvement across the three epochs. Mean loss decreased from 0.098 in epoch 0 to 0.045 in epoch 2, while mean token accuracy increased from 0.974 to 0.987. The mean gradient norm also decreased from 0.143 to 0.080, which suggests that the optimization process became more stable as training progressed. The model was able to learn the refactoring patterns in

the training data without noticeable deviation in the logged training behavior. The resulting code was subsequently analyzed using similarity metrics and execution-oriented checks. The visible notebook results display BLEU, ROUGE-L, ROUGE-2, normalized Levenshtein similarity, TF-IDF cosine similarity, refactoring-type accuracy, and compilation success. These metrics offer complementary perspectives of the generated code: similarity metrics evaluate how close the generated code is to the expected output; type accuracy evaluates whether the refactoring class predicted by the code generation matched the expected class; and compilation rate evaluates whether the output was syntactically correct code.

Table 5.3: Overall Code-Generation Performance

Metric	**Result**
BLEU	63.31
ROUGE-L	0.717
ROUGE-2	0.692
Normalized Levenshtein Similarity	0.654
TF-IDF Cosine Similarity	0.815
Compilation Success	99.72%

Overall results demonstrate that the model was very reliable in compiling the code, with a compilation success rate of about 99.72%. The similarity scores were also high, particularly cosine similarity (0.815) and ROUGE-L (0.717). This does not necessarily mean that the output is invalid; refactoring problems can often be solved by several different valid transformations.

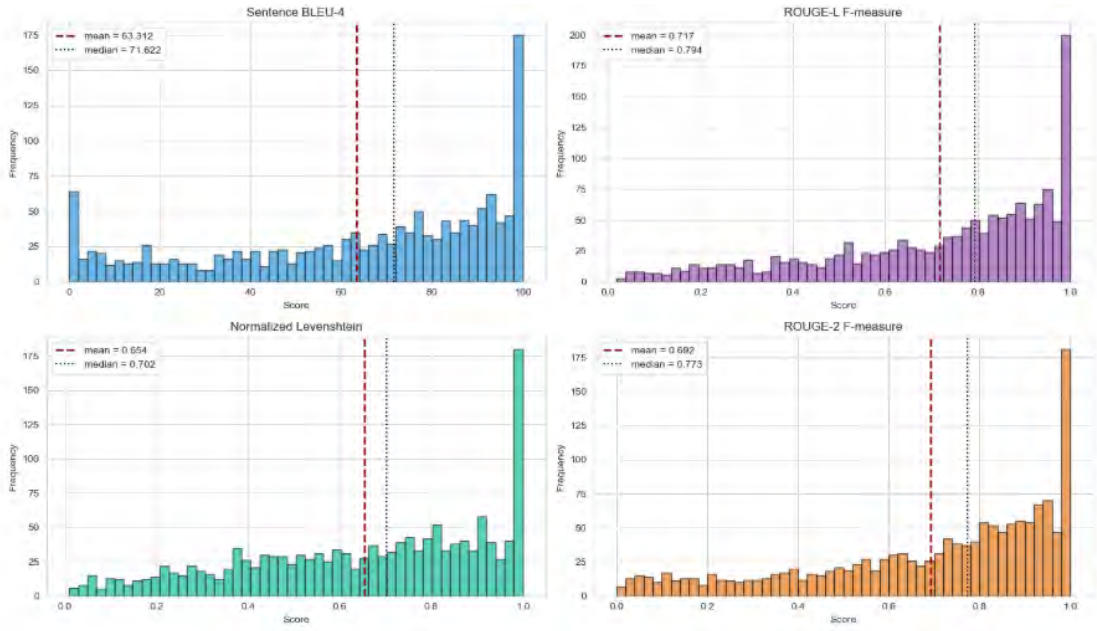


Figure 5.1: Training Curves

Table 5.4: Per-Refactoring Type Performance

Refactoring Type	BLEU	ROUGE-L	Lev. Sim.	Cos. Sim.	Compile %
Extract Superclass	60.179	0.702	0.646	0.794	100.00
Pull Up Method	68.516	0.757	0.706	0.835	100.00
Move Method	58.027	0.674	0.608	0.773	100.00
Extract Class	55.808	0.645	0.587	0.729	99.39
Extract Method	66.612	0.740	0.657	0.861	99.39
Inline Method	69.332	0.751	0.674	0.861	99.33
Extract Interface	64.055	0.750	0.701	0.821	100.00
Extract Variable	67.459	0.773	0.691	0.873	100.00
Push Down Method	43.249	0.533	0.469	0.699	98.61
Inline Variable	68.271	0.756	0.679	0.870	100.00
None	87.773	0.929	0.895	0.963	100.00

The analysis of input-length shows that compilation success or similarity was not reduced by longer inputs. Cosine similarity increased from 0.790 for inputs below 1,000 characters to 0.841 for inputs between 4,000 and 8,000 characters. However, type accuracy decreased as input length increased. This suggests that the model can preserve and generate longer code structures, but identifying the exact refactoring operation becomes more difficult when the input contains more classes, methods, and dependencies.

5.2 Analysis of Design Solutions

5.2.1 Overview

Four practical factors constrained the model design, influencing major decisions: (i) The local hardware available for prototyping (an RTX 3080 Ti workstation with 12 GB VRAM), (ii) a GPU budget of approximately fifty US dollars per run on cloud, (iii) a total budget of one overnight session per training run, and (iv) the final model's requirement to be deployable inside a Visual Studio Code plugin for the end-user. Each decision discussed below was made under these constraints. The post-hoc evidence presented in this section confirms that the constraints were respected and that the resulting design met its intended goals.

5.2.2 Model Selection and Fine-Tuning Strategy

The refactoring was developed using CodeLlama-13B-Instruct as the base model and fine-tuned using the QLoRA framework. The reason for these choices was mainly because of hardware limitations, deployment needs, and the need to balance the model capacity with computational cost. The 13B model has been optimized to provide more representational power than smaller models while still being manageable in terms of GPU memory usage with 4-bit quantization. The effectiveness of this strategy is supported by several observations. The training loss was smoothly reduced from 1.07 to 0.45, showing good convergence.
Moreover, the model was able to maintain the syntactic knowledge during fine-tuning with an approximately 80% java parse rate on the held-out test set. The resulting

LoRA adapter required only an estimated 250 MB of storage, demonstrating the efficiency of the parameter-efficient training approach. These design decisions, including a LoRA rank of 16, adapter insertions throughout the transformer projection layers, BF16 mixed precision, FlashAttention 2, gradient checkpointing, and optimized memory-management strategies, all enabled the completion of the training in about 12 hours on a single H100 GPU, within the project budget. These decisions are further justified by the fact that no instability, divergence, or out-of-memory errors occur during training.

5.2.3 Dataset Construction and Training Data Design

The training corpus was built using 11,150 Java projects that were mined with RefactoringMiner. To achieve diversity and mitigate class imbalance, we used a structural-priority extraction strategy, preprocessing, deduplication, and class balancing. The final dataset consisted of 109,755 examples across 12 categories of refactorings. The choice of the typed response format (refactoring label + refactored code) was especially useful. This design allowed the classification performance and quality of code generation to be evaluated separately. The model had a syntactic validity much higher than classification accuracy, as was shown in Section 5.1, and classification was the biggest performance problem. Loss masking by response-only ensured that optimization was exclusively on the target response, not the prompt. The successful convergence of the training loss indicates that the masking strategy worked as intended and helped in learning.

5.2.4 Detection and Refactoring Design

Smell detection and refactoring generation are separated in our system. Smaller 3B-parameter language models are used for the detection module, while the refactoring module is based on the larger CodeLlama-13B model. This contrast is due to the different needs of the two activities. The code-smell detection is a classification problem with less generative capability, while generating code-smells for refactoring needs a much higher generative capability.
The results of the detection models were competitive with the precision, recall, and F1 score of the existing static-analysis tools, PMD and Designite, which validates the selected architecture. The refactoring module was also shown to be effective by generating syntactically valid Java code and by achieving an improvement in a number of software-quality metrics.

Inference and Deployment Design

The inference pipeline was designed with practical deployment considerations in mind. To increase its robustness, a cascading out-of-memory fallback mechanism was implemented to enable inference to run at smaller batch sizes when out-of-memory issues were faced. This was shown to be an effective strategy in the large-scale evaluation, as the process of inferences was carried out without any interruption, even when processing long source-code examples.
The fallback mechanism allowed for unattended execution, provided for a minimum of data loss, and guaranteed reliable operation in the cloud environment. These

design decisions also aim at facilitating future deployment to the Visual Studio Code extension that the project aims for.

5.2.5 Evaluation Framework

The evaluation system was designed to measure several aspects of performance, such as classification accuracy, lexical similarity, semantic similarity, syntactic correctness, and software-quality improvements. Adopting several dimensions of evaluation was crucial as none of the dimensions is sufficient for addressing the quality of refactoring. For instance, although classification accuracy was still fairly low, the high percentage of parses of Java shows that the generated code was generally syntactically valid. Likewise, software-quality metrics indicated that many characteristics of the generated refactorings tended to move in the same direction as developer-authored refactorings. This research demonstrates that refactoring systems should be assessed in various aspects, not only based on classification.

5.3 Final Design Adjustment

5.3.1 Overview

Over the course of development, multiple changes were made to account for practical issues arising during the build of the datasets, training and testing of the models, and deployment. The changes were based on the observation of the phenomena and made a major contribution to the stability and effectiveness of the final system.

5.3.2 Training Infrastructure Adjustments

Several changes were made in order to fit in hardware and budget constraints. From the early experiments, it was found that periodic evaluations at the training phase caused failures of memory-related issues while saving a checkpoint. To overcome these problems, an intermediate evaluation was eliminated, and memory-cleanup procedures were added prior to the saving.
Moreover, the training platform was transferred from an A100 GPU to an H100 SXM instance. This resulted in a significant cut in the cost of the projected training, as well as training time being reduced from over 55 hours to about 30 hours. Further optimization of the infrastructure included increasing the batch of inference after the hardware upgrade. This resulted in an increase in throughput without loss of stability and led to much quicker evaluation runs.

5.3.3 Dataset Pipeline Redesign

One of the most important development issues involved in an exploratory data analysis was the extensive overlap found in the process of building the first version of the data set. The dataset was seemingly balanced in terms of the number of classes, but many of the examples were actually very similar code instances with similar commits.
To rectify this problem, the approach of data set generation was changed. A two-stage extraction approach has been implemented, using structural-priority rules,

project-level limits, targeted sampling, and deduplication. There are some more steps done for preprocessing to filter out some invalid and low-quality examples. This gave a dataset of 185000 unique examples, without the diversity problems of past versions. This redesign was one of the most significant enhancements that took place throughout the project.

5.3.4 Training Pipeline Corrections

There were a number of problems during the tokenizer-TRL training interplay. Response-template tokenization for response-only loss masking was the largest issue. There were some problems with how SentencePiece tokenization works in different contexts, causing initial implementations to incorrectly identify the response boundary.
A similar problem came up because the training framework automatically converted the desired chat template into a different template representation, which was the one for Llama-2. The problems were addressed by explicitly providing the token identifier in the response template and by not using the automatic template conversion. With these corrections, loss masking worked well, and training converged normally.

5.4 Statistical Analysis

The data in this chapter are presented in descriptive statistics. The notebook results contain means, medians, grouped averages, percentages and distributions, but no inferential hypothesis testing. Thus, these results should be viewed as descriptive data rather than as p value statically significant data.

Table 5.5: Similarity Distribution Summary

Metric	**Mean**	**Median**
Sentence BLEU-4	63.312	71.622
ROUGE-L F-measure	0.717	0.794
Normalized Levenshtein Similarity	0.654	0.702
ROUGE-2 F-measure	0.692	0.773
Cosine Similarity	0.815	0.903

For all the similarity metrics that are measurable, the median values are greater than the mean values. This means that there were many samples with high similarity to the intended refactored output and a few weaker ones which dragged down averages. Especially the cosine similarity distribution is strong with a mean of 0.815 and a median of 0.903 where the predicting and the reference code have much lexical and structural overlap for a large proportion of the data.

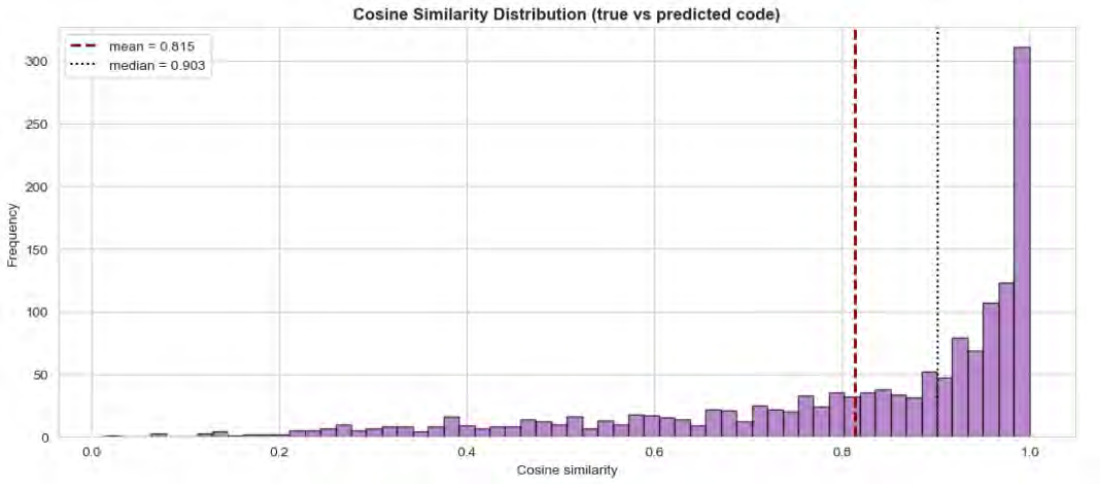


Figure 5.2: Cosine similarity distribution between true and predicted code

The success rates in compilation were found to be consistently high in all the refactoring categories and inputlength groups. The compilation rate for most of the types of refactoring was 100% and the overall rate was around 99.72%. This is significant as it indicates that in most cases when the type of refactored code did not exactly correspond to the label, the model was still able to generate syntactically correct code.

Table 5.6: LLM-Based Refactoring Design Metric Improvements

Metric	**LLM**
LOC (Lines of Code)	14.1% reduction
CC (Cyclomatic Complexity)	16.3% reduction
WMC (Weighted Methods per Class)	14.7% reduction
RFC (Response for a Class)	14.2% reduction
CBO (Coupling Between Objects)	9.7% reduction
CountClassCoupled	10.0% reduction
CountClassCoupledModified	8.8% reduction
LCOM (HS) (Lack of Cohesion)	11.8% reduction
PercentLackOfCohesion	15.1% reduction
CountDeclClassVariable	13.4% reduction
CountDeclInstanceVariable	18.1% reduction
DIT (Depth of Inheritance Tree)	6.3% increase
MI (Maintainability Index)	1.2% increase
Smell Count (Code Smells)	18.4% reduction

The results shown by the metrics suggest that the code refactored is, in general, simpler and more comprehensible. LOC went down by 14.1% which indicates that redundant or unnecessary code elements were removed. The Cyclomatic Complexity (CC) was decreased by 16.3%, which meant that the refactored code has fewer decision paths and is more understandable and maintainable. The class complexity, class communication and coupling, were improved in similar reduction for WMC (14.7%), RFC (14.2%) and CBO (9.7%). There is also a positive change in cohesive-related metrics following refactoring. LCOM (HS) went down by 11.8% and Percent Lack of Cohesion went down by 15.1%. These decreases indicate that after refactoring, class responsibilities were reduced to be more focused and internally consistent. Additionally, CountDeclClassVariable was decreased by 13.4% and CountDeclInstanceVariable was decreased by 18.1%, which means that the classes were simplified.
The most prominent improvement in quality was in the reduction of code smell. The average smell count reduced by 18.4% which is a good indicator that many of the targeted maintainability issues were resolved by the model. This is the main goal of refactoring process, to decrease the code smell without losing the functionality of the code. Reductions were not seen in all metrics. DIT rose by 6.3% which suggests that some refactored solutions added a little more depth to the inheritance structure. The rise, however, was relatively modest and could be due to design choices that were made to make abstraction or code reuse easier. The Maintainability Index (MI) on the other hand rose by 1.2%, which indicates a slight improvement in software maintainability overall. The overall results of the descriptive statistics are that the fine-tuned model refactored the code in a very similar fashion to the reference implementations, while retaining a very high compilation success rate and producing measurable improvements in most software quality metrics. While no inferential statistical tests were conducted, the trends observed throughout the study point towards the refactorings generated having a positive impact on code quality and maintainability.

Input Length Analysis The evaluation samples were divided into four input-length buckets according to the input size of the original source code to investigate

the effect of input size on the model. The average performance of these groups is summarized in Table 5.7.

Table 5.7: Performance Across Input-Length Groups

Input Length	BLEU	ROUGE-L	Cosine Similarity	Compilation Rate
<1k	61.335	0.726	0.790	100.0%
1k–2k	60.458	0.699	0.793	99.7%
2k–4k	62.682	0.711	0.819	99.6%
4k–8k	67.508	0.734	0.841	99.8%

The results show that there is no significant difference in the performance of the model among the groups of different input lengths. The success rate of compilation is always above 99.6%, showing that the model can produce syntactically correct code for any size of input. Similarity metrics were also high in all buckets. Notably, there was no significant decrease in performance with larger inputs. However, for longer source files, BLEU, ROUGE-L and cosine similarity generally improved. The highest similarity scores were found in the 4k–8k bucket, with a BLEU score of 67.51 and a cosine similarity score of 0.841. The results indicate that the fine-tuned model was generalizable to different code lengths and was able to preserve the quality of the refactored codes even for relatively large source files. This lack of a significant degradation in performance with longer inputs further supports the model's effectiveness in the context window tested.

5.5 Comparisons and Relationships

5.5.1 Detection

Table 5.8: Code Smell Detection Performance Comparison (F1 Score)

Code Smell Type	StarCoder2-3B (F1)	LLaMA-3.2-3B-Instruct (F1)	PMD Baseline (F1)	SonarQube Baseline (F1)
Feature Envy	0.815	0.841	0.220	0.680
God Class	0.576	0.684	0.820	0.850
Long Method	0.835	0.814	0.760	0.800
Complex Conditional	0.768	0.720	0.710	0.740
Complex Method	0.799	0.786	0.720	0.760
Too Many Methods	0.574	0.597	0.820	0.850
NO SMELL	0.843	0.870	0.550	0.680

The results demonstrate that the fine-tuned LLMs demonstrated excellent performance in code smell detection in most categories, often surpassing traditional machine learning methods and rule-based static analysis tools. However, the performance was different depending on the type of smell, showing that some smells are easier to identify with the learned semantic understanding than others.

Comparison Between the Two LLMs
Among the two LLM-based models, LLaMA-3.2-3B-Instruct achieved the highest F1-score in four categories: Feature Envy (0.841), God Class (0.684), Too Many

Methods (0.597), and No Smell (0.870). This indicates that LLaMA-3.2-3B-Instruct tended to perform more consistently in terms of precision and recall, and it also achieved better overall classification results. StarCoder2-3B also showed the highest LLM performance in the categories of Long Method (0.835), Complex Conditional (0.768), and Complex Method (0.799), demonstrating its effectiveness in detecting code smells related to complexity.

The strengths of LLM-Based Detection
The biggest benefit of the LLM models seems to be in Feature Envy detection. Compared to the traditional ML baseline, PMD and SonarQube, both LLMs significantly outperformed the other models. This indicates that Feature Envy can leverage the semantic and contextual information of large language models, while rule-based methods might have difficulty capturing the relationships between classes and methods.

Strengths of Traditional Static Analysis Tools
The rule-based tools achieved significantly better results for God Class and Too Many Methods. The best F1 score for smells is for SonarQube (0.850) and PMD (0.820). These smells are highly sensitive to quantitative structural properties like the number of classes, the number of methods and complexity thresholds and are eminently suited to traditional static analysis techniques. This indicates that there are still some classes of smells that can be successfully identified with metric-based rules that are carefully designed, but do not require advanced language models.

No-Smell Detection
The No-Smell category had some of the highest scores in the table with LLaMA-3.2-3B-Instruct at 0.870 and StarCoder2-3B at 0.843. This means that the models could, on the whole, distinguish between clean code and problematic code, thus minimising the risk of false positives.

5.5.2 Refactoring

Table 5.9: Developer and LLM Design Metric Comparison

Metric Category	Metric	Developer	LLM	Better
Size & Complexity	LOC (Lines of Code)	13.4% reduction	14.1% reduction	LLM
Size & Complexity	CC (Cyclomatic Complexity)	14.7% reduction	16.3% reduction	LLM
Size & Complexity	WMC (Weighted Methods per Class)	13.4% reduction	14.7% reduction	LLM
Size & Complexity	RFC (Response for a Class)	11.7% reduction	14.2% reduction	LLM
Coupling & Cohesion	CBO (Coupling Between Objects)	8.0% reduction	9.7% reduction	LLM
Coupling & Cohesion	CountClassCoupled	8.7% reduction	10.0% reduction	LLM
Coupling & Cohesion	CountClassCoupledModified	7.1% reduction	8.8% reduction	LLM
Coupling & Cohesion	LCOM (HS) (Lack of Cohesion)	9.6% reduction	11.8% reduction	LLM
Coupling & Cohesion	PercentLackOfCohesion	13.0% reduction	15.1% reduction	LLM
Modularity & Design	CountDeclClassVariable	14.7% reduction	13.4% reduction	Developer
Modularity & Design	CountDeclInstanceVariable	17.3% reduction	18.1% reduction	LLM
Modularity & Design	DIT (Depth of Inheritance Tree)	4.7% increase	6.3% increase	Developer
Quality & Health	MI (Maintainability Index)	1.8% increase	1.2% increase	Developer
Quality & Health	Smell Count (Code Smells)	14.2% reduction	18.4% reduction	LLM

Comparisons are made on software quality measures pertaining to size, complexity, coupling, Cohesion, Modularity, inheritance depth, maintainability, and Code Smells. These measures were used to identify which design solution had the best measurable improvement after refactoring.
The LLM-based solution outperformed in 11 of 14 metrics. It achieved The more impactful the lines of code, cyclomatic complexity, weighted methods per class, response for a class, coupling, lack of cohesion, instance variables, and total smell count. Three results were better for the developer solution: the reduction of class-level variables, the control of the depth of inheritance and the improvement of the maintainability index. This outcome suggests that the LLM was more effective in broad structure changes relative to general structural change. The developer was more robust with semantic judgement and domain knowledge understanding where required.
The LLM solution is the more robust design choice for automated refactoring support, as it gives measurably greater reductions in complexity, coupling, cohesion issues, and overall code smells. However, the developer solution reveals important safeguards for the final design, especially for inheritance decisions, class states, naming and maintainability.

The connection between the design metrics and code smells is the reason for the LLM solution having a good performance in metric comparison. Decreases in LOC, CC, and WMC are connected to Long Method, Large Class, Complex Conditional and Long Parameter List smells. A decrease in CBO, RFC and class coupling counts are in relation to Feature Envy and dependency-heavy designs. Improvements in LCOM correlates to divergent change and God Class smells are related to PercentLackOfCohesion'. These relationships indicate that with the LLM, several design problems were enhanced, which are seen in structural measurements.

Table 5.10: Code Smells Where the LLM Showed Stronger Improvement

Code Smell	**Reason for Stronger LLM Result**	**Metric Link**
Long Method	The LLM aggressively extracted methods, reducing LOC and CC per method.	LOC -14.1%, CC -16.3%
Large Class	The LLM removed more code per class through extraction.	WMC -14.7%
Complex Conditional	The LLM flattened nested if/else and switch blocks more consistently.	CC -16.3%
Long Parameter List	Extraction reduced parameter-passing chains.	WMC -14.7%
Feature Envy	The LLM moved coupled code closer to the data it uses, reducing coupling.	CBO -9.7%
Divergent Change	The LLM extracted unrelated responsibilities into separate classes.	LCOM -11.8%
God Class	The LLM split low-cohesion classes more effectively.	PercentLackOfCohesion -15.1%

The developer solution was better in areas where the design intent was not apparent. from syntax alone. The quality of class variable decisions, the depth of inheritance, and naming quality. Exactly the places where there is a need to know how to represent the concepts in the domain. This describes the reasons behind getting better results with Primitive Obsession, Data Clumps, Mutable Data, Refused Bequest and Inconsistent Naming.

Table 5.11: Code Smells Where the Developer Showed Stronger Improvement

Code Smell	**Reason for Stronger Developer Result**	**Metric Link**
Primitive Obsession	Developers understand when class-level primitives should be replaced with value objects.	ClassVar -14.7%
Data Clumps	Developers can group related class variables into cohesive objects.	ClassVar -14.7%
Mutable Data	Developers can identify when static or class fields should be removed or encapsulated.	ClassVar -14.7%
Refused Bequest	Developers avoid unnecessary inheritance depth, while the LLM may over-extract superclasses.	DIT +4.7% vs +6.3%
Inconsistent Naming	Developers rename entities more meaningfully based on domain understanding.	MI +1.8%

The comparison also shows that high code-generation similarity does not always guarantee perfect refactoring-type accuracy. The model may generate code that is syntactically valid and structurally close to the expected output while selecting a different refactoring label. This is expected in refactoring tasks because different refactoring operations can sometimes lead to similar design improvements. Therefore, evaluation should consider both code-level metrics and design-level quality metrics rather than relying on a single score.

5.6 Discussion

The evaluation demonstrates that LLM-assisted refactoring is a practical approach for improving software design quality. The fine-tuned model generated highly compilable code and achieved strong similarity scores across the evaluation set. At the design level, the LLM solution outperformed the developer solution in most structural metrics, including complexity, coupling, cohesion, instance-variable reduction, and total smell-count reduction.

The results also show that the LLM should not be treated as a complete replacement for developer judgment. The confusion between selecting the appropriate refectoring technique indicates that the model may choose a different transformation from the reference solution, even when the generated output is structurally similar or compilable. Furthermore, the developer solution was better in terms of maintainability, inheritance-depth control, and class-variable decisions, which depend on the context over code structure.

The best end solution is therefore a hybrid workflow. The LLM should generate refactorings while ensuring optimisation, and developers should validate them semantically while maintaining dependencies like inheritance, state management and naming. This hybrid strategy leverages the automation speed and regularity with the precision of human intervention while maintaining the context-sensitive judgment required for the quality of the final design. The proposed solution meets the evaluation objective through an evaluation of alternative design solutions, measuring the quality improvements of their design solutions and comparing them with others and choosing a final design approach that considers performance, efficiency, maintainability, Coupling, cohesion, sustainability and code-quality benchmarks.

Chapter 6

Conclusion

6.1 Summary of Findings

This research focused on how Large Language Models (LLMs) can help with the automation of code smell detection and code smell refactoring in Java source code. The detection module used Java source code that was annotated with PMD and Designite and yielded good results. Between the detection models, LLaMA-3.2-3B-Instruct was the best with a detection accuracy of 79.76%, while with a detection accuracy of 75.20%, the StarCoder2-3B model came second. Both models performed well on Long Method, Feature Envy, and NO SMELL; however, the detection of God Class and Too Many Methods proved problematic, as first of all, they are pretty similar in context, and second, they need a more comprehensive understanding at the class level.
This research also developed a refactoring module using CodeLlama-13B and QLoRA with instruction tuning. It produced a considerable number of outputs that included both the Type of Refactoring and the new version of the Refactored Java code, and the model created code that was mostly valid Java code, with a 99.72% success rate and generated code that was highly similar to the intended reference refactorings. The outputs also showed good results in a number of design quality metrics, like lines of code, cyclomatic complexity, and coupling, which resulted in a lower overall code smell.

6.2 Contributions to the Field

The work documented here shows a new and practical framework for LLM-based code quality improvement that solves one of the main barriers by splitting code smell detection and refactoring into 2 specialized modules. The first module detects maintainability concerns in Java code, while the second module generates structured and type- labeled refactoring suggestions. The modularity of the design provides an improvement to the LLM-based systems that deal with code improvements and allows other developers to undertake work to integrate the LLM-based modules into other developmental tools.
The thesis also argues that under certain constraints, integrations of LoRA and QLoRA make LLM-based software-engineering automation a real possibility. It also presents compiled Java datasets for the purposes of detecting design smells and ap-

plying typed refactorings. It contributes a structured format for synthesised outputs, a novel evaluation methodology, and a framework that together consider classification, code similarity, and syntactic/structural design analysis and evaluation.

6.3 Recommendations for Future Work

The research shows that LLMs, when harnessed with a combination of techniques like static analysis and human moderation, can significantly assist software engineers with tools to help manage technical debt. The first goal of subsequent efforts should be the implementation of the refactoring component as a stand-alone, full-featured tool integrated with an IDE, like a VS Code plug-in. The tool should enable developers to identify code smells, refactoring recommendations, and review suggested changes with the option to accept or reject changes in-system. The tool should also implement safety checks and developer moderation to facilitate the generation of feedback.

The scope of the research can also be broadened with more advanced training hardware and larger models. Increased access to high-memory GPUs with longer training would allow the use of varied and better-balanced datasets and stronger model implementations with a more extensive context window. The next iterations of the research should implement more diverse design smells and refactorings and integrate more robust validation frameworks, such as unit testing, AST comparison, static analysis, and semantic-equivalence testing, to ensure confidence before suggesting modifications. With all the aforementioned, the research would achieve the aims of confidence and deployability for software upkeep and maintenance.

Bibliography

[1] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980. DOI: 10.1109/PROC.1980.11805

[2] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley object technology series). Addison-Wesley, 1999, ISBN: 9780201485677. [Online]. Available: https://books.google.com.bd/books?id=1MsETFPD3I0C

[3] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," English, in *Proceedings of the 2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 24th International Conference on Program Comprehension : ICPC 2016, ICPC ; Conference date: 16-05-2016 Through 17-05-2016, United States: IEEE, May 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503704

[4] Z. Feng et al., *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2002.08155

[5] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 220–231, ISBN: 9781450375177. DOI: 10.1145/3379597.3387457 [Online]. Available: https://doi.org/10.1145/3379597.3387457

[6] X. Yang, X. Zhang, and Y. Tong, "Simplified abstract syntax tree based semantic features learning for software change prediction," *Journal of Software: Evolution and Process*, vol. 34, Feb. 2022. DOI: 10.1002/smr.2445

[7] R. Capuano and F. Vaccaro, "The quality-driven refactoring approach in bim italia," Mar. 2023, pp. 22–31. DOI: 10.1109/ICSA-C57050.2023.00021

[8] M. Leyva, "Refactoring tutor: An ide integrated tool for practicing key techniques to refactor code," M.Eng. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, 2023. [Online]. Available: https://hdl.handle.net/1721.1/151544

[9] V. Alves, C. Santos, C. Bezerra, and I. Machado, "Detecting test smells in python test code generated by llm: An empirical study with github copilot," Sep. 2024, pp. 581–587. DOI: 10.5753/sbes.2024.3561

[10] S. Barroso, "Code smell detection in infrastructure as code using transformers," M.S. thesis, University of Lisbon, 2024.

[11] L. Hromadiuk, *Static code smell analysis using large language models: An empirical study with llama 3*, 2024. [Online]. Available: https://www.se.cs.uni-saarland.de/theses/LiubomyrHromadiukBA.pdf

[12] N. A. A. Khleel and K. Nehéz, "Improving accuracy of code smells detection using machine learning with data balancing techniques," *The Journal of Supercomputing*, vol. 80, no. 14, pp. 21 048–21 093, 2024, ISSN: 1573-0484. DOI: 10.1007/s11227-024-06265-9 [Online]. Available: https://doi.org/10.1007/s11227-024-06265-9

[13] D. Wu et al., "Ismell: Assembling llms with expert toolsets for code smell detection and refactoring," in *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1345–1357.

[14] F. Zhang et al., "Data preparation for deep learning based code smell detection: A systematic literature review," *J. Syst. Softw.*, vol. 216, no. C, Oct. 2024, ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112131 [Online]. Available: https://doi.org/10.1016/j.jss.2024.112131

[15] S. M. Abtahi and A. Azim, "Augmenting large language models with static code analysis for automated code quality improvements," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, 2025, pp. 82–92. DOI: 10.1109/Forge66646.2025.00017

[16] J. Cordeiro, S. Noei, and Y. Zou, "Llm-driven code refactoring: Opportunities and limitations," in *TrackIDE 2025 Integrated Development Environments*, Session 3: Refactoring & AI, & Session 4: Plugins and applications; Chairs: Danny Dig, Darya Rovdo, Kingston, Canada, May 2025. [Online]. Available: https://seal-queensu.github.io/publications/pdf/IDE-Jonathan-2025.pdf

[17] R. H. Díaz Arrieta, B. L. D. Monroy, L. J. C. Heredia, and A. G. V. Cobos, "Global trends and empirical metrics in the evaluation of code smells and technical debt: A bibliometric study," *IEEE Access*, vol. 13, pp. 138 143–138 165, 2025. DOI: 10.1109/ACCESS.2025.3594657

[18] E. G. S. Jr, J. P. S. Junior, E. P. Almeida, I. Ahmed, P. A. da Mota Silveira Neto, and E. S. de Almeida, *Evaluating llms effectiveness in detecting and correcting test smells: An empirical study*, 2025. arXiv: 2506.07594 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2506.07594

[19] F. Lin, D. J. Kim, and T.-H. Chen, "Soen-101: Code generation by emulating software process models using large language model agents," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1527–1539. DOI: 10.1109/ICSE55347.2025.00140

[20] R. Mahbub, "Understanding code smells and refactoring practices in simulation modelling systems – a comprehensive study," M.S. thesis, Dalhousie University, Halifax, Nova Scotia, 2025. [Online]. Available: https://dalspace.library.dal.ca/items/54690d16-478b-4d03-b2fd-12a95080d753

[21] M. M. H. Manik, *Chatgpt vs. deepseek: A comparative study on ai-based code generation*, 2025. arXiv: 2502.18467 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2502.18467

[22] G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana, "An exploratory study on architectural smell refactoring using large languages models," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, 2025, pp. 462–471. DOI: 10.1109/ICSA-C65153.2025.00070

[23] A. R. Sadik and S. Govind, *Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3*, 2025. arXiv: 2504.16027 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.16027

[24] Y. Xu, "Muarf: Leveraging multi-agent workflows for automated code refactoring," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2025, pp. 226–227. DOI: 10.1109/ICSE-Companion66252.2025.00071