# CodeFume: LLM-Powered Detection and Refactoring of Object-Oriented Code Smells

Alara Zeybek*, Ece Beyhan†, Yaşar Tatlıcıoğlu‡, Serhat Yılmaz§
Group T9, CS437
*22102544, †22003503, ‡22003856, §22002537
{alara.zeybek,ece.beyhan,yasar.tatlicioglu,serhaty}@ug.bilkent.edu.tr

***Abstract*—CodeFume is an LLM-driven platform for detecting and refactoring object-oriented code smells in software systems. Existing static analyzers often miss semantic context and fail to provide actionable refactorings; CodeFume overcomes these limitations through a two-stage, instruction-based pipeline that leverages large language models (LLMs) for nuanced smell identification and automated remediation. First, CodeFume issues a structured detection prompt to an LLM to identify four prevalent smells—Long Method, Large Class, Feature Envy, and God Object—returning type, location, severity, and descriptive justifications in JSON format. Second, a dedicated refactoring prompt generates targeted code transformations grounded in SOLID principles, and an automated equivalence check verifies functional correctness. We evaluate CodeFume on 100 manually labeled Java cases and 54 real-world C# samples, measuring detection accuracy, refactoring quality, and structural code metrics. Our results show detection rates of 100 % for Long Method, 89 % for Large Class, and 52.5 % for Feature Envy; refactorings achieve up to 94 % functional equivalence on class-level smells and deliver up to 76 % cyclomatic complexity reduction and 212 % maintainability improvement for structural issues. A comparative study of Google Gemini versus GPT-4.1-nano highlights Gemini's superior maintainability preservation and token efficiency. Key contributions include: (i) a novel prompt-based detection and refactoring architecture; (ii) structured JSON outputs for seamless integration; (iii) an end-to-end evaluation framework; and (iv) comparative LLM analysis. By combining semantic understanding with traditional metrics, CodeFume enables developers to rapidly identify and remediate design flaws, reducing technical debt and improving long-term code quality.**

## I. Introduction

Modern software systems often suffer from structural issues commonly known as *code smells* that reduce readability, maintainability, and long-term quality. Traditional static analysis tools like SonarQube can detect surface-level patterns, but they struggle with semantic context, offer limited customization, and rarely provide actionable refactorings.

To overcome these limitations, we introduce **CodeFume** [1], a web-based assistant powered by Large Language Models (LLMs) for semantic smell detection and automated refactoring. CodeFume employs a prompt-driven architecture consisting of two stages: detection and refactoring.

In the detection stage, CodeFume sends code snippets to an LLM using a detailed system prompt to identify smells and return structured JSON output. In the refactoring stage, a second LLM uses this analysis to produce targeted code changes while minimizing hallucinations. All prompts are optimized for zero-shot instruction clarity and token efficiency to ensure consistent and interpretable outputs.

Our key contributions are:

- A novel prompt-based pipeline for semantic code smell detection.
- An automated LLM-driven refactoring engine producing SOLID-aligned improvements.
- A comprehensive evaluation on Java and C# datasets, measuring accuracy, equivalence, and code-quality metrics.
- A comparative analysis of Gemini vs. GPT-4.1-nano, highlighting trade-offs in maintainability and token efficiency.

Beyond its practical utility as a developer tool, CodeFume is the result of a structured research effort aimed at evaluating the feasibility and reliability of prompt-based code intelligence. This study investigates its effectiveness in generating accurate, high-quality code refactorings compared to both baseline tools and human-written alternatives. The following research questions guide our analysis:

### A. Research Questions

**RQ1:** How reliably can *CodeFume* detect complex code smells (god class, feature envy, long method, large class)?

**RQ2:** To what extent can *CodeFume*'s LLM-generated refactorings be considered functionally equivalent to human-written refactorings?

**RQ3:** How does the quality of refactored code produced by *CodeFume* compare to human refactorings in terms of maintainability and complexity metrics?

**RQ4:** What are the limitations of prompt-based refactoring in capturing nuanced, context-specific human suggestions?

**RQ5:** How effective is *CodeFume* at reducing structural code complexity according to standard static analysis metrics?

**RQ6:** How do different LLMs (GPT vs. Gemini) affect the quality, consistency, and correctness of code smell detection and refactoring in *CodeFume*?

## II. Background and Related Work

Traditional tools for code smell detection, such as **SonarQube** [2] and **JDeodorant** [3], rely on static analysis techniques and hand-crafted rules applied over abstract syntax trees (ASTs). These approaches are effective at identifying well-known structural patterns such as *Long Method* or *God*

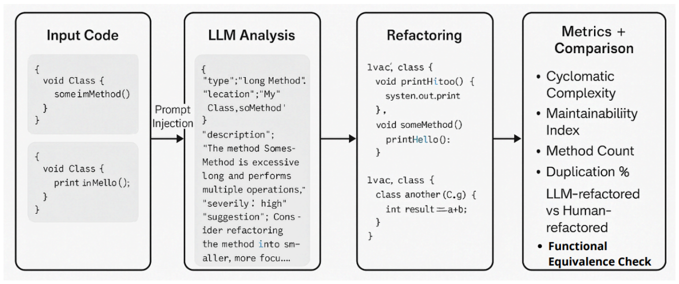
*Object*, but they often fall short in understanding the semantic context of code. This limitation becomes particularly evident in non-standard, obfuscated, or dynamically typed codebases.

Recent work has explored using large language models (LLMs) for code synthesis, completion, and transformation tasks [4], [5], but few systems leverage LLMs for comprehensive smell detection, explanation, and automated refactoring. While tools like GitHub Copilot [6] offer assistive code suggestions, they do not explicitly identify design flaws or recommend structural improvements grounded in software engineering principles.

**CodeFume** addresses this gap by introducing an LLM-driven architecture for semantic code smell analysis. Unlike rule-based tools, CodeFume applies prompt-based reasoning using Gemini to semantically detect smells, generate refactorings, and conduct comparative analyses of functionality and structure. This three-stage pipeline enhances flexibility and explainability, offering a practical solution to maintainability issues often overlooked by traditional tools.

## III. System Overview: CodeFume

**CodeFume** is an intelligent code analysis platform designed to detect and remediate code smells—subtle structural issues that suggest deeper design flaws or maintainability problems in software systems. The platform provides an end-to-end workflow that begins with user-submitted source code, either via direct text input or file upload, and proceeds through a multi-stage backend pipeline for automated analysis and refactoring.



```
{
  void Class {
    someimMethod()
  }
}
```

```
{
  void Class {
    print inMello();
  }
}
```

```
{
"type";"long Method".
"lecation";"My"
 Class,soMethod'
}
"description";
"The method Somes-
Method is excessive
long and performs
multiple operations,"
"severily: high"
"suggestion"; Cons-
ider refactoring
the method into sm-
aller, more focu....
```

```
lvac, class {
  void printHitoo() {
    systen.out.print
  },
  void someMethod()
    printHello():
  }

lvac, class {
  class another (C.g) {
    int result =a+b;
  }
}
```

Fig. 1: System architecture

CodeFume features a modern architecture, with a React-based frontend for user interaction and a Flask-powered backend for logic orchestration. The core of its analytical capabilities is powered by the Gemini large language model, which is integrated at multiple critical stages to enable high-fidelity semantic analysis and transformation of code.

The analytical engine of CodeFume functions in three sequential stages:

1) **Code Smell Detection:** A dedicated prompt is issued to the Gemini model to identify prevalent code smells such as *Long Method*, *Large Class*, *Feature Envy*, and *God Object*. Detected smells are annotated with severity levels (e.g., High, Medium), descriptive justifications, and relevant code snippets.
2) **Refactoring:** A separate prompt generates an improved version of the code that addresses the specific issues previously detected. These refactorings are aligned with established software engineering principles such as the Single Responsibility Principle and encapsulation.
3) **Comparative Analysis:** A final prompt compares the original and refactored code to assess structural and functional differences. This includes a summary of lines added and removed, a functional equivalence check to ensure correctness, and a side-by-side diff with explanations of key changes.

The output interface presents users with a structured, visually coherent display of the results. It details what code smells were detected, their severity and impact, and provides targeted refactoring suggestions. By combining the reasoning capabilities of large language models with the discipline of static analysis, CodeFume offers a novel, explainable, and practical tool to improve code maintainability, reduce technical debt, and promote clean architecture in software development.

## IV. Methodology

### A. Testing System Overview

Our experimental framework consists of a custom pipeline built in Python that integrates LLM-based code smell detection and automated refactoring. The core logic is implemented in two key scripts:

- `automated_test.py` – conducts end-to-end experiments on Long Method, Feature Envy, and God Object smells using static datasets.
- `large_class_refactor.py` – processes live GitHub code samples for Large Class smell using extracted links.

The main configuration is handled via a custom service class, `CodeSmellService`, which manages LLM communication (detection, refactoring, comparison), JSON parsing, and error handling.

### B. Smell Detection and Refactoring

We used Google's Gemini model for both smell detection and code improvement. Two separate prompts are utilized:

- **Detection Prompt:** Identifies only the four smells (Long Method, Large Class, Feature Envy, God Object), returning type, location, severity, description, and refactoring suggestion in JSON format.
- **Refactoring Prompt:** Uses the above JSON output and the original code to return refactored files with changes limited to resolving the detected smells.

All results are processed using zero-shot prompting to avoid overfitting and maintain generalizability.

### C. Dataset

The system is evaluated over two types of datasets:

- **Java Dataset (Static, Offline) [7]:**
  - 100 manually labeled cases.

  - Code smells: Long Method, Feature Envy, God Object.
  - File structure: `before.java` (original), `after.java` (human refactored).
- **C# Dataset (Online, GitHub-based) [8]:**
  - Extracted using using GitHub API.
  - Used only for Large Class detection and refactoring.

### D. Evaluation Metrics

We implemented a custom static analysis engine in Python to compute the following metrics:

- **Cyclomatic Complexity (CC)**: Total number of control flow branches (if, for, while, switch) + 1.
- **Method Count**: Number of function declarations.
- **Average Method Length**: Ratio of total lines to method count.
- **Maintainability Index (MI)**: Calculated using Halstead metrics, CC, and total lines.
- **Comment Density**: Ratio of comment lines to total lines (comments + code).
- **Duplication Percentage**: Percentage of repeated line hashes in the code.
- **Total Lines**: Code + comment lines.

Metric computation is done for:

- The original (smelly) code.
- Human-refactored code.
- LLM-refactored code.

### E. Functional Equivalence Evaluation

To assess whether LLM refactorings preserved functional behavior, we used a third prompt to compare human and LLM refactorings and return:

- Structural differences.
- Overlap in suggestions applied.
- Final verdict: `true` / `false` on equivalence.

This step ensures that we do not overestimate quality purely based on metric improvements.

### F. Output Format

All results for each test case are written to `analysis_results.txt` which includes:

- Original and refactored code
- Diff summary
- Smell analysis in JSON
- Functional equivalence result
- Metric breakdown for each version

Thus for each test case, the system generates a structured result file containing the following components:

- **Code Smell Detection Output:** JSON-formatted analysis including smell type, location, severity, description, and suggested refactorings.
- **LLM Refactoring Output:** Code generated by the LLM based on the detected smells and suggestions.
- **Comparison Report:** LLM-generated comparison between human and model refactorings to evaluate structural changes and functional equivalence.
- **Metric Reports:** Quantitative evaluation of code quality before and after refactoring using complexity, maintainability, and duplication-related metrics.
- **Aggregated Results:** Final summaries of smell detection counts, equivalence rates, and metric averages are compiled into a consolidated result file.

These outputs provide a comprehensive basis for both qualitative and quantitative assessment of LLM-based code refactoring performance, and are further discussed in the Results and Discussion section.

### G. LLM Comparison Experiment

To investigate how different LLMs affect the performance of CodeFume, we conducted a controlled comparison using two leading models: OpenAI's `gpt-4.1-nano` and Google's `gemini-2.0-flash`. Both models were integrated into the same experimental pipeline, using identical code samples, prompts, and evaluation procedures to ensure fairness and consistency.

For each selected input, both LLMs were given the same detection and refactoring prompts. The outputs were then evaluated using the same set of metrics: detection accuracy, structural code quality improvements (e.g., complexity reduction, maintainability gain), and functional equivalence against human refactorings (if available).

This dual-model setup allowed us to analyze the strengths and weaknesses of each LLM in the context of semantic code analysis and transformation. The results of this comparison are discussed in Section V, highlighting both overlapping capabilities and distinct behavioral patterns across models.

## V. Results

Our comprehensive analysis of CodeFume's performance across four distinct code smell types reveals significant insights into LLM-based detection and refactoring capabilities. We evaluated CodeFume on two datasets: 100 manually labeled Java cases (static) and 54 real-world C# samples (based on GitHub). During initial trials, we experimented with five different prompt formulations per smell but found the accuracy unsatisfactory. We then consolidated our approach into a single, comprehensive prompt that detects all four code smells in one pass—outputting results in a uniform JSON schema—to maximize LLM performance and deliver a consistent, user-friendly experience. Since each sample in our data sets was preannotated with its true code smells, we were able to directly measure the detection accuracy of the LLM and compare the automated refactorings of Gemini with human-crafted versions. The detailed results are shown below.

### A. Detection Accuracy

This section presents results for RQ1. CodeFume demonstrated varying detection accuracy across different code smell types:

- Long Method: 100% detection rate
- Large Class: 89% detection rate (48/54 cases correctly identified)
- Feature Envy: 52.5% detection rate
- God Object: Detection varied based on classification approach (26% when classified strictly as God Object, 74% when categorized as Large Class)

TABLE I: Detection Accuracy by Code Smell

| Smell Type | Detection Rate (%) |
|---|---|
| Long Method | 100.0 |
| Large Class | 89.0 |
| Feature Envy | 52.5 |
| God Object (strict) | 26.0 |
| God Object (as Large Class) | 74.0 |

This variance suggests that LLMs have stronger pattern recognition capabilities for structural smells with clear syntactic signatures (Long Method, Large Class) compared to semantic-dependent smells like Feature Envy that require deeper contextual understanding. Our prompt-based approach was most effective for smells with distinct structural patterns and clear boundaries.

### *B. Functional Equivalence*

This section presents results for RQ2. Functional equivalence between LLM-generated and human refactorings varied by smell type:

- Long Method: 60% functional equivalence
- Feature Envy: 59% functional equivalence
- God Object: 59% functional equivalence
- Large Class: 94.4% functional equivalence (51/54 cases)

TABLE II: Functional Equivalence of LLM Refactorings

| Smell Type | Equivalence Rate (%) |
|---|---|
| Long Method | 60.0 |
| Feature Envy | 59.0 |
| God Object | 59.0 |
| Large Class | 94.4 |

The high equivalence for Large Class refactorings suggests that LLMs have a strong understanding of class decomposition principles. The lower equivalence rates for other smell types indicate challenges in preserving complex behavioral semantics, particularly when refactoring methods with intricate control flows and variable interdependencies.

### *C. Refactoring Quality by Smell Type*

This section presents results for RQ3.

#### *1) Long Method:*

- Human refactorings reduced complexity by 46.7%, while Gemini showed no improvement
- Human refactorings improved maintainability by 18.2%, while Gemini decreased it by 9.8%
- Human refactorings reduced method length, while Gemini increased it by 44.7%
- Human refactorings reduced code size, while Gemini increased it by 23.6%

Listing 1: Before Refactoring (Long Method)

```java
public void processOrder(Order o) {
   // 45 lines of mixed logic...
   // calculation, validation,
   // notification, persistence
}
```

Listing 2: After Refactoring (Long Method)

```java
// Refactored by CodeFume
public void processOrder(Order o) {
   validateEach(o);
   calculateSum(o);
   notifyPerson(o);
   persistOrder(o);
}
private void validateEach(Order o) {...}
private void calculateSum(Order o) {...}
private void notifyPerson(Order o) {...}
private void persistOrder(Order o) {...}
```

Fig. 2: Long Method refactor example produced by CodeFume

#### *2) Feature Envy:*

- Human refactorings reduced complexity by 61%, while Gemini achieved only 8.4% reduction
- Human refactorings improved maintainability by 38.1%, while Gemini achieved 16.9%
- Gemini achieved better method length reduction (70.5%)
- Human refactorings reduced code size by 56.8%, while Gemini achieved only 1.9% reduction
- Gemini created 5x more methods through extraction

#### *3) God Object:*

- Gemini achieved superior complexity reduction compared to humans (75.8% vs. 28.8%)
- Gemini showed 5x greater maintainability improvement than human approaches
- Gemini reduced method length by 65%, while human refactorings increased it by 22%
- Gemini reduced code size by 83.7%, outperforming human refactorings

#### *4) Large Class:*

- Gemini achieved substantial complexity reduction (61.2%)
- Gemini showed remarkable maintainability improvement (212.2%)
- Gemini reduced code size by 64.1%

These results reveal that CodeFume outperforms human refactorings for structural smells (God Object, Large Class) but underperforms for behavioral smells (Long Method, Feature Envy).

### D. Limitations of Prompt-Based Refactoring

This section presents results for RQ4. Our analysis identified several limitations in prompt-based refactoring:

- **Algorithm Understanding Gap**: LLMs struggle to fully comprehend the algorithmic intent behind complex methods, leading to functionally correct but suboptimal refactorings.
- **Context Sensitivity**: Prompt-based approaches have limited ability to capture domain-specific considerations that humans incorporate in their refactorings, such as performance trade-offs or architectural coherence.
- **Execution Flow Limitations**: LLMs showed difficulty in tracing complex execution paths, particularly when refactoring methods with multiple conditional branches or exception handling.
- **Variable Lifecycle Blindness**: Refactorings often missed opportunities to optimize variable scopes or failed to identify redundant state management that human refactorings addressed.
- **Documentation and Naming Context**: While LLMs could restructure code, they sometimes lost important context in method and class naming conventions established in the original codebase.

These limitations were most evident in Feature Envy refactorings, where the LLM often identified the presence of the smell but struggled to move functionality to the correct recipient class based on semantic understanding of the code's purpose.

### E. Structural Complexity Reduction

This section presents results for RQ5. CodeFume's effectiveness at reducing structural complexity varied significantly by smell type:

*1) Cyclomatic Complexity Reduction:*

- Long Method: No improvement (0%)
- Feature Envy: Modest reduction (8.4%)
- God Object: Substantial reduction (75.8%)
- Large Class: Significant reduction (61.2%)

TABLE III: Cyclomatic Complexity Reduction by Smell Type

| Smell Type | Reduction (%) |
|---|---|
| Long Method | 0.0 |
| Feature Envy | 8.4 |
| God Object | 75.8 |
| Large Class | 61.2 |

*2) Maintainability Index Improvement:*

- Long Method: Degradation (-9.8%)
- Feature Envy: Moderate improvement (16.9%)
- God Object: Substantial improvement (outperforming humans 5x)
- Large Class: Dramatic improvement (212.2%)

TABLE IV: Maintainability Index Improvement by Smell Type

| Smell Type | Improvement (%) |
|---|---|
| Long Method | -9.8 |
| Feature Envy | 16.9 |
| God Object | 5x more than human |
| Large Class | 212.2 |

*3) Method Length and Code Size:*

- Long Method: Increased method length (44.7%) and code size (23.6%)
- Feature Envy: Reduced method length (70.5%) but minimal code size reduction (1.9%)
- God Object: Significantly reduced method length (65%) and code size (83.7%)
- Large Class: Reduced code size (64.1%)

These metrics demonstrate that CodeFume is highly effective at reducing structural complexity for class-level smells (God Object, Large Class) but less effective for method-level smells (Long Method, Feature Envy).

### F. Model Comparison: Gemini vs. GPT

This section presents results for RQ6. Our comparative analysis between Gemini and GPT-4 revealed important differences:

*1) Detection Accuracy:*

- Long Method: Similar performance (Gemini: 100%, GPT: 99%)
- Feature Envy: GPT showed significantly better detection (92.9% vs. 52.5%)

*2) Functional Equivalence:*

- Long Method: Comparable (Gemini: 60%, GPT: 58%)
- Feature Envy: GPT slightly higher (64.6% vs. 59.6%)

*3) Quality Metrics:*

- Long Method:
  - Complexity reduction: GPT better (2.32 vs. 0.28)
  - Maintainability: Gemini better (+0.73 vs. -4.51)
- Feature Envy:
  - Complexity: Gemini reduced (1.42) while GPT increased (-1.06)
  - Maintainability: Gemini improved (+5.86) while GPT degraded (-5.38)

*4) Refactoring Approaches:*

- GPT employed more aggressive decomposition with better complexity reduction but sometimes at the cost of maintainability
- Gemini used a more balanced approach that better preserved code readability and maintainability
- GPT created more methods during extraction, while Gemini focused on meaningful reorganization

*5) Token Efficiency:*

- Gemini offers significantly higher token limits (1,000,000 vs. 8,000-16,000)

- Gemini can process approximately 80,000-100,000 lines of Java code vs. 700-1,400 for GPT

Based on these findings, we selected Gemini for our final implementation due to its better maintainability preservation, more balanced refactoring approach, and significantly higher token limits that enable processing larger codebases.

### *G. Structural vs. Behavioral Refactoring*

Our findings highlight a clear pattern: LLMs excel at structural refactoring (God Object, Large Class) but struggle with behavioral refactoring (Long Method, Feature Envy):

*1) LLM Strengths in Structural Refactoring:*

- Pattern recognition of class-level architectural patterns
- Clear boundaries for decomposition
- Mechanical transformation through class extraction
- Lower risk of breaking functionality during structural changes

*2) LLM Limitations in Behavioral Refactoring:*

- Algorithm understanding gaps
- Context sensitivity requirements
- Performance trade-offs blindness
- Execution flow limitations
- Variable lifecycle understanding challenges

## VI. Discussion

Our evaluation revealed clear patterns and several practical takeaways:

We found that CodeFume excels at detecting and refactoring structural code smells (Long Method, Large Class, God Object), achieving up to 100% detection accuracy and over 200% improvements in maintainability. In contrast, behavioral smells (Feature Envy, method-level issues) consistently saw lower detection (nearly 50%) and smaller metric gains. It was surprising that for God Object, Gemini outperformed human refactorings in both complexity reduction (75.8% vs. 28.8%) and maintainability (5× improvement).

### *A. Key Insights*

- Structural Patterns Are "Easy Wins." LLMs leverage clear syntactic boundaries to break apart large classes and god objects reliably.
- Behavioral Semantics Remain Challenging. Without domain context, the model struggles to reassign responsibilities (Feature Envy) or decompose complex methods meaningfully.
- Prompt Design Matters. Our switch to a single, unified prompt vastly improved consistency and user experience.

### *B. Strengths*

- High detection accuracy on class-level smells.
- Automated refactorings for structural smells preserve functionality at 94 % equivalence.
- JSON-structured outputs integrate seamlessly into CI/CD pipelines and IDE plugins.

### *C. Limitations*

Although our "Limitations" subsection (RQ4) detailed algorithmic and context-sensitivity gaps, we also note:

- Dataset Scope: Only Java and C# were tested; results may not generalize to dynamic languages or functional paradigms.
- Prompt Variability: Even our final prompt can yield different outputs under minor wording changes.
- No Human-in-the-Loop Study: We have yet to measure developer acceptance or time savings in practice.

### *D. Applicability*

CodeFume is best suited for:

- *Legacy System Modernization:* Rapidly decomposing monolithic classes in large codebases.
- *Automated Code Review:* As a CI check to flag and optionally auto-refactor structural smells.
- *Educational Tools:* Teaching students about SOLID principles through hands-on refactoring examples.

## VII. Threats to Validity

- **Internal Validity:** The smell annotations in our datasets may contain bias or inconsistencies, potentially affecting detection accuracy metrics.
- **External Validity:** Our study is limited to Java and C# codebases, and findings may not generalize to other programming languages or paradigms.
- **Construct Validity:** The metrics we used to evaluate code quality (cyclomatic complexity, maintainability index) may not fully capture the nuanced aspects of code quality and readability.
- **Reliability:** LLM outputs show variability based on prompt formulation and internal randomness, so results may fluctuate across repeated runs.

## VIII. Conclusion and Future Work

### *A. Contributions*

We summarize our key contributions:

1) A novel two-stage, prompt-based LLM pipeline for semantic code-smell detection and automated refactoring.
2) A uniform JSON schema output enabling CI/CD and IDE integration.
3) A comprehensive evaluation framework on Java and C# datasets measuring detection accuracy, functional equivalence, and code-quality metrics.
4) A comparative analysis of Google Gemini vs. GPT-4.1-nano, highlighting trade-offs in maintainability and token efficiency.

### *B. Conclusion*

CodeFume demonstrates that LLM-driven approaches offer a promising direction for automated code smell detection and refactoring, with several important findings:

First, our analysis shows that LLMs possess strong capabilities in identifying structural code smells through zero-shot prompting, with accuracy rates ranging from 89-100% for Long Method and Large Class. However, detection accuracy drops significantly (52.5%) for more semantically complex smells like Feature Envy that require deeper contextual understanding.

Second, preserving functionality during refactoring remains challenging, with equivalence rates of 59-60% for most smell types. The exception is Large Class refactoring, where equivalence reached 94% - suggesting LLMs better understand class decomposition than method-level transformations.

Third, LLM effectiveness varies significantly by smell type. For structural refactorings (God Object, Large Class), LLMs can achieve dramatic improvements in complexity (61-76% reduction) and maintainability (up to 212% improvement) that often exceed human refactorings. However, for behavioral refactorings (Long Method, Feature Envy), LLMs struggle to match human quality, sometimes even degrading metrics.

Fourth, zero-shot instruction-based prompting showed limitations in capturing context-specific considerations, algorithm understanding, and execution flow analysis that human experts incorporate in their refactorings.

Fifth, CodeFume demonstrated highly effective structural complexity reduction for class-level smells (61-76% reduction) but less effectiveness for method-level smells.

Finally, different LLMs showed distinct strengths, with Gemini offering better maintainability preservation and token efficiency, while GPT provided more aggressive decomposition and feature envy detection.

These results confirm that LLM-based code smell detection and refactoring is viable, particularly for structural code smells. While not yet matching human refactoring quality across all smell types, tools like CodeFume can serve as valuable assistants in the refactoring process, especially for initial decomposition of large classes and god objects.

### *C. Future Work*

Several promising directions for future research emerge from our findings:

1) **Few-shot Prompting:** Investigating whether adding examples of high-quality refactorings to prompts could improve LLM performance, particularly for behavioral refactorings like Long Method and Feature Envy.
2) **Domain-specific Fine-tuning:** Training specialized models on code refactoring datasets could potentially improve both detection accuracy and refactoring quality.
3) **Hybrid Approaches:** Combining traditional static analysis tools with LLM-based semantic understanding could leverage the strengths of both approaches.
4) **Expanded Smell Coverage:** Extending CodeFume to detect additional code smells like Duplicate Code, Primitive Obsession, and Data Class would increase the tool's utility.
5) **Integration with IDEs:** Developing IDE plugins for CodeFume would streamline the workflow for developers, allowing real-time smell detection and suggested refactorings.
6) **Human-LLM Collaboration Models:** Exploring interactive workflows where LLMs suggest refactorings and humans approve or modify them could optimize the refactoring process.
7) **Performance Optimization:** Investigating techniques to reduce token usage and execution time would make CodeFume more practical for real-world development scenarios.
8) **Cross-language Support:** Extending support to additional programming languages would broaden the tool's applicability.

As LLM capabilities continue to evolve, tools like CodeFume have significant potential to enhance code quality by making refactoring more accessible and efficient. The combination of semantic understanding and structured transformation offered by LLMs represents a powerful new paradigm for automated code improvement that complements traditional static analysis approaches.

## References

[1] CodeFume, "Codefume," n.d., accessed: 2025-05-16. [Online]. Available: https://github.com/CS437-24-SP/T9-CodeFumeRepo

[2] SonarSource, "Static code analysis using sonarqube: A step-by-step guide," n.d., accessed: 2025-05-16. [Online]. Available: https://www.sonarsource.com/learn/static-code-analysis-using-sonarqube/

[3] N. Tsantalis, "Jdeodorant," n.d., accessed: 2025-05-16. [Online]. Available: https://github.com/tsantalis/JDeodorant

[4] OpenAI, "Openai codex," 2021, accessed: 2025-05-16. [Online]. Available: https://en.wikipedia.org/wiki/OpenAI_Codex

[5] Microsoft Research, "Codebert: A pre-trained model for programming and natural languages," 2020, accessed: 2025-05-16. [Online]. Available: https://github.com/microsoft/CodeBERT

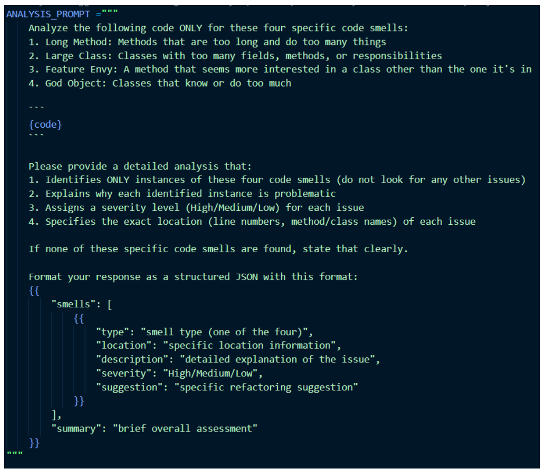
[6] GitHub, "Getting code suggestions in your ide with github copilot," n.d., accessed: 2025-05-16. [Online]. Available: https://docs.github.com/en/copilot/using-github-copilot/getting-code-suggestions-in-your-ide-with-github-copilot

[7] Kachanov, Vladimir and Markov, Sergey, "Trusted code smells dataset," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7612725

[8] Pereira dos Reis, José and Brito e Abreu, Fernando and Figueiredo Carneiro, Glauco, "Code smells dataset (oracles)," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6555241

## Appendix A
## Appendix

Example prompts, outputs, and full metric tables can be found in our replication package.



~~~~
ANALYSIS_PROMPT ="""
    Analyze the following code ONLY for these four specific code smells:
    1. Long Method: Methods that are too long and do too many things
    2. Large Class: Classes with too many fields, methods, or responsibilities
    3. Feature Envy: A method that seems more interested in a class other than the one it's in
    4. God Object: Classes that know or do too much

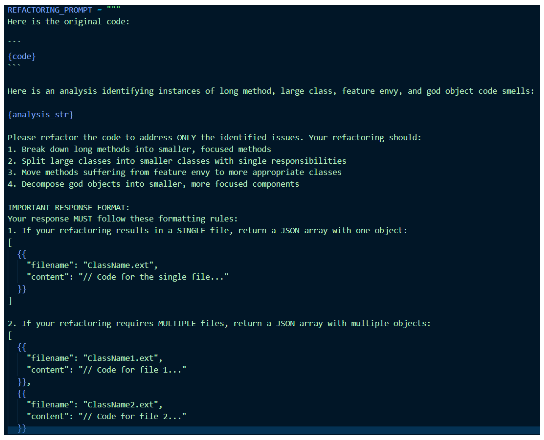
    ```
    {code}
    ```

    Please provide a detailed analysis that:
    1. Identifies ONLY instances of these four code smells (do not look for any other issues)
    2. Explains why each identified instance is problematic
    3. Assigns a severity level (High/Medium/Low) for each issue
    4. Specifies the exact location (line numbers, method/class names) of each issue

    If none of these specific code smells are found, state that clearly.

    Format your response as a structured JSON with this format:
    {{
        "smells": [
            {{
                "type": "smell type (one of the four)",
                "location": "specific location information",
                "description": "detailed explanation of the issue",
                "severity": "High/Medium/Low",
                "suggestion": "specific refactoring suggestion"
            }}
        ],
        "summary": "brief overall assessment"
    }}
"""
~~~~

Fig. 3: Example detection prompt used in our experiments.



~~~~
REFACTORING_PROMPT = """
Here is the original code:

```
{code}
```

Here is an analysis identifying instances of long method, large class, feature envy, and god object code smells:

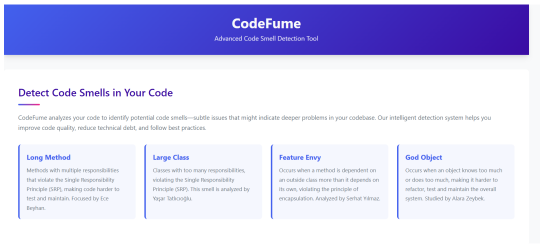
{analysis_str}

Please refactor the code to address ONLY the identified issues. Your refactoring should:
1. Break down long methods into smaller, focused methods
2. Split large classes into smaller classes with single responsibilities
3. Move methods suffering from feature envy to more appropriate classes
4. Decompose god objects into smaller, more focused components

IMPORTANT RESPONSE FORMAT:
Your response MUST follow these formatting rules:
1. If your refactoring results in a SINGLE file, return a JSON array with one object:
[
  {{
    "filename": "ClassName.ext",
    "content": "// Code for the single file..."
  }}
]

2. If your refactoring requires MULTIPLE files, return a JSON array with multiple objects:
[
  {{
    "filename": "ClassName1.ext",
    "content": "// Code for file 1..."
  }},
  {{
    "filename": "ClassName2.ext",
    "content": "// Code for file 2..."
  }}
~~~~

Fig. 4: Example refactoring prompt used in our experiments.



**CodeFume**

Advanced Code Smell Detection Tool

**Detect Code Smells in Your Code**

CodeFume analyzes your code to identify potential code smells—subtle issues that might indicate deeper problems in your codebase. Our intelligent detection system helps you improve code quality, reduce technical debt, and follow best practices.

**Long Method**
Methods with multiple responsibilities that violate the Single Responsibility Principle (SRP), making code harder to test and maintain. Focused by Ece Beyhan.

**Large Class**
Classes with too many responsibilities, violating the Single Responsibility Principle (SRP). This smell is analyzed by Yaşar Tatlıcıoğlu.

**Feature Envy**
Occurs when a method is dependent on an outside class more than it depends on its own, violating the principle of encapsulation. Analyzed by Serhat Yılmaz.

**God Object**
Occurs when an object knows too much or does too much, making it harder to refactor, test and maintain the overall system. Studied by Alara Zeybek.

Fig. 5: CodeFume main page.

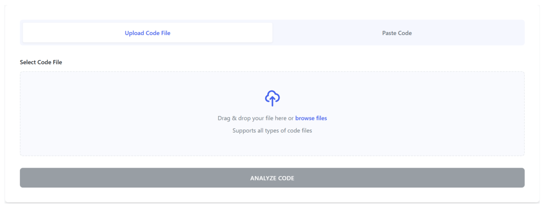
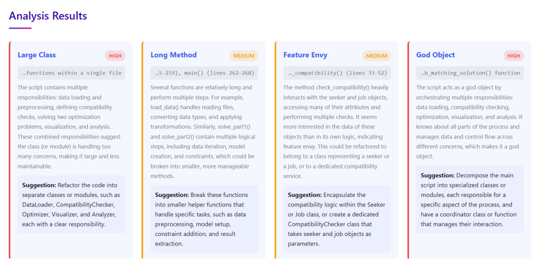

Upload Code File
Paste Code
Select Code File
Drag & drop your file here or browse files
Supports all types of code files
ANALYZE CODE

Fig. 6: Input taking page.



**Analysis Results**

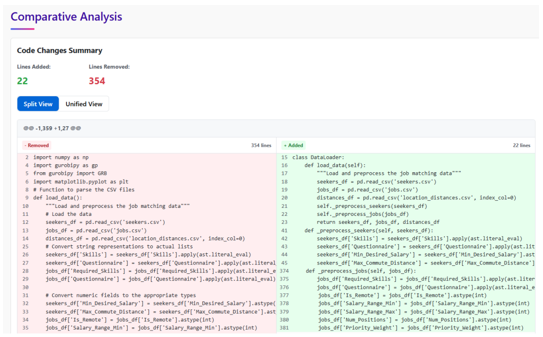
**Large Class** HIGH
`…functions within a single file`
The script contains multiple responsibilities: data loading and preprocessing, defining compatibility checks, solving two optimization problems, visualization, and analysis. These combined responsibilities suggest the class (or module) is handling too many concerns, making it large and less maintainable.
**Suggestion:** Refactor the code into separate classes or modules, such as DataLoader, CompatibilityChecker, Optimizer, Visualizer, and Analyzer, each with a clear responsibility.

**Long Method** MEDIUM
`…5-259), main() (lines 262-268)`
Several functions are relatively long and perform multiple steps. For example, load_data() handles reading files, converting data types, and applying transformations. Similarly, solve_part1() and solve_part2() contain multiple logical steps, including data iteration, model creation, and constraints, which could be broken into smaller, more manageable methods.
**Suggestion:** Break these functions into smaller helper functions that handle specific tasks, such as data preprocessing, model setup, constraint addition, and result extraction.

**Feature Envy** MEDIUM
`…_compatibility() (lines 33-52)`
The method check_compatibility() heavily interacts with the seeker and job objects, accessing many of their attributes and performing multiple checks. It seems more interested in the data of these objects than in its own logic, indicating feature envy. This could be refactored to belong to a class representing a seeker or a job, or to a dedicated compatibility service.
**Suggestion:** Encapsulate the compatibility logic within the Seeker or Job class, or create a dedicated CompatibilityChecker class that takes seeker and job objects as parameters.

**God Object** HIGH
`…b_matching_solution() function`
The script acts as a god object by orchestrating multiple responsibilities: data loading, compatibility checking, optimization, visualization, and analysis. It knows about all parts of the process and manages data and control flow across different concerns, which makes it a god object.
**Suggestion:** Decompose the main script into specialized classes or modules, each responsible for a specific aspect of the process, and have a coordinator class or function that manages their interaction.

Fig. 7: Sample analysis result output.



**Comparative Analysis**

**Code Changes Summary**

Lines Added: 22
Lines Removed: 354

Split View | Unified View

@@ -1,359 +1,27 @@

- Removed — 354 lines

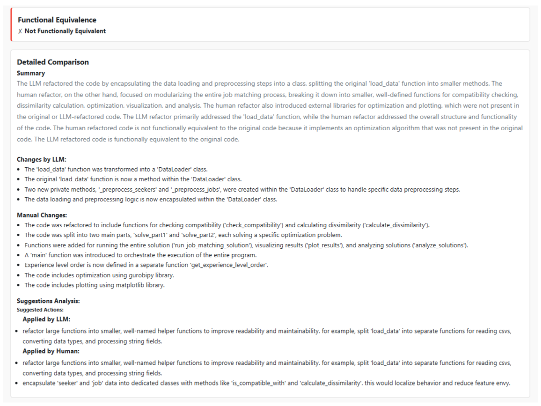
```
import numpy as np
import gurobipy as gp
from gurobipy import GRB
import matplotlib.pyplot as plt
# Function to parse the CSV files
def load_data():
    """Load and preprocess the job matching data"""
    # Load the data
    seekers_df = pd.read_csv('seekers.csv')
    jobs_df = pd.read_csv('jobs.csv')
    distances_df = pd.read_csv('location_distances.csv', index_col=0)
    # Convert string representations to actual lists
    seekers_df['Skills'] = seekers_df['Skills'].apply(ast.literal_eval)
    seekers_df['Questionnaire'] = seekers_df['Questionnaire'].apply(ast.literal
    jobs_df['Required_Skills'] = jobs_df['Required_Skills'].apply(ast.literal_e
    jobs_df['Questionnaire'] = jobs_df['Questionnaire'].apply(ast.literal_eval)

    # Convert numeric fields to the appropriate types
    seekers_df['Min_Desired_Salary'] = seekers_df['Min_Desired_Salary'].astype(
    seekers_df['Max_Commute_Distance'] = seekers_df['Max_Commute_Distance'].ast
    jobs_df['Is_Remote'] = jobs_df['Is_Remote'].astype(int)
    jobs_df['Salary_Range_Min'] = jobs_df['Salary_Range_Min'].astype(int)
```

+ Added — 22 lines

```
class DataLoader:
    def load_data(self):
        """Load and preprocess the job matching data"""
        seekers_df = pd.read_csv('seekers.csv')
        jobs_df = pd.read_csv('jobs.csv')
        distances_df = pd.read_csv('location_distances.csv', index_col=0)
        self._preprocess_seekers(seekers_df)
        self._preprocess_jobs(jobs_df)
        return seekers_df, jobs_df, distances_df
    def _preprocess_seekers(self, seekers_df):
        seekers_df['Skills'] = seekers_df['Skills'].apply(ast.literal_eval)
        seekers_df['Questionnaire'] = seekers_df['Questionnaire'].apply(ast.lit
        seekers_df['Min_Desired_Salary'] = seekers_df['Min_Desired_Salary'].ast
        seekers_df['Max_Commute_Distance'] = seekers_df['Max_Commute_Distance']
     def _preprocess_jobs(self, jobs_df):
        jobs_df['Required_Skills'] = jobs_df['Required_Skills'].apply(ast.liter
        jobs_df['Questionnaire'] = jobs_df['Questionnaire'].apply(ast.literal_e
         jobs_df['Is_Remote'] = jobs_df['Is_Remote'].astype(int)
         jobs_df['Salary_Range_Min'] = jobs_df['Salary_Range_Min'].astype(int)
         jobs_df['Salary_Range_Max'] = jobs_df['Salary_Range_Max'].astype(int)
         jobs_df['Num_Positions'] = jobs_df['Num_Positions'].astype(int)
         jobs_df['Priority_Weight'] = jobs_df['Priority_Weight'].astype(int)
```

Fig. 8: Sample comparative analysis output.



**Functional Equivalence**
✗ **Not Functionally Equivalent**

**Detailed Comparison**

**Summary**

The LLM refactored the code by encapsulating the data loading and preprocessing steps into a class, splitting the original 'load_data' function into smaller methods. The human refactor, on the other hand, focused on modularizing the entire job matching process, breaking it down into smaller, well-defined functions for compatibility checking, dissimilarity calculation, optimization, visualization, and analysis. The human refactor also introduced external libraries for optimization and plotting, which were not present in the original or LLM-refactored code. The LLM refactor primarily addressed the 'load_data' function, while the human refactor addressed the overall structure and functionality of the code. The human refactored code is not functionally equivalent to the original code because it implements an optimization algorithm that was not present in the original code. The LLM refactored code is functionally equivalent to the original code.

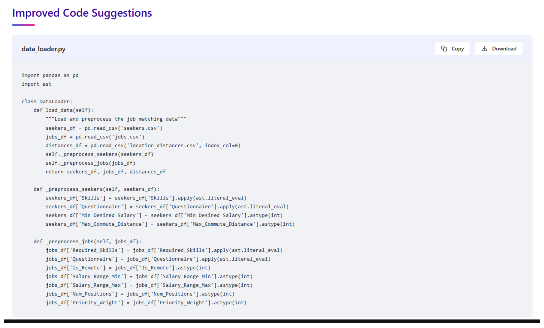
**Changes by LLM:**
- The 'load_data' function was transformed into a 'DataLoader' class.
- The original 'load_data' function is now a method within the 'DataLoader' class.
- Two new private methods, '_preprocess_seekers' and '_preprocess_jobs', were created within the 'DataLoader' class to handle specific data preprocessing steps.
- The data loading and preprocessing logic is now encapsulated within the 'DataLoader' class.

**Manual Changes:**
- The code was refactored to include functions for checking compatibility ('check_compatibility') and calculating dissimilarity ('calculate_dissimilarity').
- The code was split into two main parts, 'solve_part1' and 'solve_part2', each solving a specific optimization problem.
- Functions were added for running the entire solution ('run_job_matching_solution'), visualizing results ('plot_results'), and analyzing solutions ('analyze_solutions').
- A 'main' function was introduced to orchestrate the execution of the entire program.
- Experience level order is now defined in a separate function 'get_experience_level_order'.
- The code includes optimization using gurobipy library.
- The code includes plotting using matplotlib library.

**Suggestions Analysis:**
**Suggested Actions:**
**Applied by LLM:**
- refactor large functions into smaller, well-named helper functions to improve readability and maintainability. for example, split 'load_data' into separate functions for reading csvs, converting data types, and processing string fields.

**Applied by Human:**
- refactor large functions into smaller, well-named helper functions to improve readability and maintainability. for example, split 'load_data' into separate functions for reading csvs, converting data types, and processing string fields.
- encapsulate 'seeker' and 'job' data into dedicated classes with methods like 'is_compatible_with' and 'calculate_dissimilarity'. this would localize behavior and reduce feature envy.

Fig. 9: Sample comparative explanation output.



**Improved Code Suggestions**

data_loader.py

```
import pandas as pd
import ast

class DataLoader:
    def load_data(self):
        """Load and preprocess the job matching data"""
        seekers_df = pd.read_csv('seekers.csv')
        jobs_df = pd.read_csv('jobs.csv')
        distances_df = pd.read_csv('location_distances.csv', index_col=0)
        self._preprocess_seekers(seekers_df)
        self._preprocess_jobs(jobs_df)
        return seekers_df, jobs_df, distances_df

    def _preprocess_seekers(self, seekers_df):
        seekers_df['Skills'] = seekers_df['Skills'].apply(ast.literal_eval)
        seekers_df['Questionnaire'] = seekers_df['Questionnaire'].apply(ast.literal_eval)
        seekers_df['Min_Desired_Salary'] = seekers_df['Min_Desired_Salary'].astype(int)
        seekers_df['Max_Commute_Distance'] = seekers_df['Max_Commute_Distance'].astype(int)

    def _preprocess_jobs(self, jobs_df):
        jobs_df['Required_Skills'] = jobs_df['Required_Skills'].apply(ast.literal_eval)
        jobs_df['Questionnaire'] = jobs_df['Questionnaire'].apply(ast.literal_eval)
        jobs_df['Is_Remote'] = jobs_df['Is_Remote'].astype(int)
        jobs_df['Salary_Range_Min'] = jobs_df['Salary_Range_Min'].astype(int)
        jobs_df['Salary_Range_Max'] = jobs_df['Salary_Range_Max'].astype(int)
        jobs_df['Num_Positions'] = jobs_df['Num_Positions'].astype(int)
        jobs_df['Priority_Weight'] = jobs_df['Priority_Weight'].astype(int)
```

Fig. 10: Sample improved code suggestion.