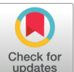
Ingénierie des Systèmes d'Information
Vol. 31, No. 3, March, 2026, pp. 879-888
Journal homepage: http://iieta.org/journals/isi

CodeSense: A Static Analysis Tool for Detecting Code Smells



Hesham Adnan Alabbasi[1*], Mohanad Ridha Ghanim[1], Hiba A. Abu-Alsaad[2]

[1] Computer Science Department, College of Education, Mustansiriyah University, Baghdad 10052, Iraq
[2] Computer Engineering Department, College of Engineering, Mustansiriyah University, Baghdad 10052, Iraq

Corresponding Author Email: hesham.alabbasi@uomustansiriyah.edu.iq

Copyright: ©2026 The authors. This article is published by IIETA and is licensed under the CC BY 4.0 license (http://creativecommons.org/licenses/by/4.0/).

https://doi.org/10.18280/isi.310319

Received: 15 November 2025
Revised: 25 January 2026
Accepted: 18 March 2026
Available online: 31 March 2026

Keywords:
hybrid detection, software testing, code smells, context-aware analysis, program analysis, static analysis tool

ABSTRACT

Code smells are indicators of the design problems that affect the quality and maintainability of the software. Traditional tools that rely on static analysis can detect rule-based violations, but not complex ones. Alternative sophisticated tools that integrate machine learning (ML) can be computationally intensive and need to develop a model for a specific code smell. This paper proposes a new static analysis tool, called CodeSense, to address these challenges, describing the design, concept, implementation and evaluation of this tool. CodeSense integrates program analysis techniques with a unified detection approach, with a goal of achieving context-aware detection of code smells. It optimizes the current tools by improving the detection of design problems in the early stages, enhancing the software quality and decreasing the requirement for an extensive refactoring process. The empirical results for CodeSense show competitive results in accurately detecting code smells; it achieves higher F1-scores (0.87) for all the code smells compared to the baseline tools: SonarQube, Programming Mistake Detector (PMD) and Checkstyle, which are 0.75, 0.71 and 0.66. Our research demonstrates the need for more intelligent and integrated static analysis tools to meet the demands of today's software development.

1. INTRODUCTION

Software engineering topics range from the basics of computer science, such as data structures, algorithms and programming languages. Its focus today is on software design and development. Software development has to deal with changing software requirements, complex systems, real-time constraints, and security [1, 2]. In addition, the need for maintainable, robust, and clean code is more important than ever. The growing number of programmers with varying expertise calls for tools to help enforce code quality.

Although great progress has been made in software engineering, Code smell tools are still difficult to develop. Code smells are not necessarily bugs (bug reports contain all crucial information that helps software development to identify and resolve problems occurring in software) [3] or functional errors, but features or signals in program code that indicate potential problems negatively affect software technical debt [4]. They should be fixed to prevent low quality of software, reduce maintainability and cause technical bugs [5]; therefore, several approaches using Machine learning (ML) have been developed to identify and mitigate code smells [4].

Conventional static analysis tools have been the major means of solving coding problems, with a typical approach that applies predefined rules and metrics to detect common anti-patterns. These tools are generally good for many simple violations; however, they fail to understand the complex contextual relation within a codebase. The inability to comprehend context relationships within a codebase leads to a large amount of false positives and fails to detect important code smells. Moreover, the latest static analysis tools, which use ML techniques, as marked by reference [6], are a critical sub-discipline in the multidisciplinary fields of Artificial Intelligence (AI) and computer science [7]. Software engineering can integrate ML and AI [8].

ML technology can greatly enhance the intelligence of equipment, which can improve production efficiency, decrease energy cost and enhance product quality [9]. ML holds great promise for code smell detection, but also has limitations in terms of computational cost. These involve the need to train individual models for each code smell, which makes their use and scalability difficult.

Thanks to the latest developments in Large Language Models (LLMs) and Generative AI (GenAI) technologies, software development has improved its capabilities in code generation as well as analysis. LLMs can process programming and natural languages, opening a new opportunity to address the challenges in existing code smell detection. Moreover, LLMs can detect complex code patterns that traditional rule-based and metric-based approaches cannot, and discover new or previously unknown code smells.

In this design, we assumed that a static analysis tool with advanced contextual understanding would help to improve recall and precision in code smell detection, especially for subtle and complex code smells that are difficult to detect through conventional approaches. This could provide more context-sensitive refactoring recommendations, in turn

reducing the manual work needed to enhance the code quality.

Our paper makes the following contributions:

- A novel static analysis tool to enhance code smell detection.
- The tool, named CodeSense, combines the accuracy of conventional static analysis with advanced contextual knowledge provided by modern AI technologies.
- We show that through integrating such advantages, CodeSense can be more accurate than existing tools.

The rest of this paper is structured as follows: Section 2 provides background on code smells and discusses related work in static analysis and code quality using AI. Section 3 describes the design and technical aspects of CodeSense, such as the architecture of the system and the algorithms used. Section 4 provides further implementation details, including tools used and challenges faced. Section 5 describes the experimental framework and performance evaluation, including the metrics, benchmarks and comparison standards. Section 6 reflects on the significance of our results, limitations, and possible enhancements. Finally, Section 7 concludes the paper and discusses future work.

2. BACKGROUND

In this section, we introduce code smells understanding, survey the current state of the art approaches and tools for code smell detection and place our suggested method within the current state of the art.

2.1 Code smells: Definition, types, and impact

Code smells are characteristics in source code that, although not necessarily bugs, suggest underlying issues with the design or implementation of the system. Refactoring expert Martin Fowler identified different code smells, each a red flag that a code might be in need of restructuring [10]. These smells are important as they affect maintainability, readability, and extendibility of the software, which in turn leads to a higher technical debt. We can group common code smells as outlined in the study [10] into:

Bloaters: These smells are related to code, methods, or classes that have become too large or complicated, making them more difficult to comprehend and to manage. The list includes: (1) Long Method, a method with too many lines of code, making it difficult to understand and reuse, (2) Large Class, a class that is trying to handle too many responsibilities, violating the Single Responsibility Principle (SRP), (3) Primitive Obsession, using primitive data types for domain concepts that warrant dedicated objects, (4) Long Parameter List, a method with an too much parameters, complicating its usage, and (5) Data Clumps, groups of data that extremely appear together, indicating the requirement for new object.

Object-Orientation Abusers: These smells are due to misapplication or incomplete application of object-oriented programming principles. This includes: (1) Switch statements are often listed when used to differentiate behavior based on object type that could be handled better by polymorphism, (2) Temporary fields are instance variables used only in certain circumstances, which can lead to confusion about the object's state, (3) Refused Bequest refers to a subclass inheriting unnecessary functionality from its superclass, specifying poor inheritance hierarchy, and (4) Alternative Classes with different mediators highlight two classes that perform similar functions but lack a standard interface, thereby reducing reusability.

Change Preventers: These smells cause the codebase to be difficult to change, and a single logical change needs a large number of changes. This group includes: (1) Divergent Change occurring when a single change modified for multiple unrelated reasons, making it weak, (2) Parallel Inheritance Hierarchies requires creating subclass in single hierarchy as a subclass is added in another, which causes an increase in the maintenance overhead, (3) Shotgun Surgery describing a situation where one change requires many small changes in many different classes, making updates error prone.

Dispensable: These are elements that don't add value and can be eliminated to simplify the code. This category includes: (1) Duplicate Code, where identical code blocks appearing in multiple locations, increasing maintenance effort and introducing inconsistencies, (2) Comments can be a code smell when they are redundant, outdated, excessive, masking underlying design problems, (3) Data Class is a class that primarily holds data with minimal behavior, breaking encapsulation, (4) Dead Code unreachable or unexecuted code which clutters the codebase, (5) Lazy Class describing a class which does too little in justifying its existence, add unnecessary complexity, and (6) Speculative Generality involves creating code for the anticipated future needs which never materialize, resulting in unnecessary abstraction.

Code smells (especially Long Parameter List and Complex Method) are very common in actual projects, and they have a large impact on the quality of software. There is a requirement for code smells detection in order to enhance maintainability of the software and to increase productivity of the developers, thereby lowering the cost of software development.

2.2 Related works

Code smell analysis has gone to the next stage of being automated. It started with manual reviews that were time-consuming and simple rule-based checks that were subjective and prone to human error. These limitations were overcome by new automated and metric-based techniques. They are also used to compute different software measures, including Cyclomatic Complexity, Lack of Cohesion of Methods (LCOM) [11], and Depth of Inheritance Tree (DIT) [12], and compare them to predetermined values to detect possible code smells.

Popular examples consist of: SonarQube (An extremely powerful platform which has a significant role in the quality and maintainability of software projects) [13], Programming Mistake Detector (PMD) (An extensible multilanguage static code analyzer) [14], and Checkstyle (a development tool that can help programmers write Java code to a coding standard that give comprehensive reports on code quality issues) [15], whereas efficient for various structural smells, a major limitation regarding metric-based methods is the nonuniformity as well as subjectivity of such thresholds, which frequently need manual tuning and interpretation by experienced developers [16].

Later, more recently, code smell detection has used ML techniques to exploit their capacity to learn complex patterns based on data and is less dependent on fixed thresholds. Various ML algorithms, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and autoencoders, were employed to enhance the accuracy of detection. A rule-based classification technique in the study

[13] refactors content, and multi-linear regression (MLR) modeling assesses the use of the battery with software measurements of smelly code instances. The methods based on ML demonstrate a potential high accuracy, yet they have significant drawbacks, including the large size of the labeled datasets required, the computational cost of training a unique model on each code smell, and the challenges in interpreting the decisions of the models.

The advent of LLMs has brought about new possibilities for the analysis of code. LLMs, like Llama [17], have an impressive capacity to read and produce human-like text, and more and more code. Their ability to analyze natural and programming languages supports a more contextual sense of code, and they may be able to identify more subtle and more complex code smells that may otherwise be overlooked by traditional static analyzers. Recent studies have started to investigate the use of LLMs in detecting code smells, and some studies have shown a competitive performance, especially with structural code smells. Nonetheless, the existing results indicate that, although LLMs can be useful supplementary tools, they are not prepared to substitute existing static analysis tools, and their overall performance might not be substantially different compared to the traditional ones.

Table 1 provides a comparative summary of existing code smell detection techniques, categorized by their methodological approach. As shown, most prior works rely solely on automated rule-based or metric-based methods, while hybrid approaches that combine multiple techniques remain limited. Our proposed tool, CodeSense, adopts a hybrid strategy that integrates rule-based, pattern-based, and heuristic-based detection to address the limitations identified in the literature.

Table 1. Comparison among code smell techniques

Ref. No.	Approach				Description
	Only Automated	**Only Metric-Based**	**Hybrid (Automated and Metric-Based)**	**AI / LLMs**	
[11]		√			Uses inter-smell relations and maintainability metrics in product lines.
[12]		√			Classic CK proposed by Chidamber and Kemerer (hereafter CK) metrics suite (WMC, DIT, CBO, RFC, LCOM, NOC).
[13]	√				SonarQube platform, automated detection of smells via metrics + rules.
[14]	√				Static code analyzer; detects smells/issues automatically.
[15]	√				Rule-based tool to enforce coding standards and detect smells.
[16]		√			Code smell analysis in Android apps, linked to performance/battery consumption.
[17]				√	Train language models for optimal performance at different inference budgets by using more tokens than usual.
This work (Code Sense)			√		Hybrid analyzer tool designed to improve the detection of code smells.

2.3 Research gaps

Although the code smell detection techniques have evolved, there are still a few gaps in existing practices critical to our proposed tool of analyzing code, which are as follows:

- Limited contextual understanding: Conventional rule-based and metric-based tools tend to work with localized patterns of code, and do not have a global view of the architecture and semantic ties of the codebase. This drawback leads to false positives or even inability to detect smells.
- Computational cost of advanced ML models: Although ML provides good pattern recognition, most of the models of detecting code smell are computationally intensive, especially when they need to be trained individually on each type of smell. This has an adverse impact on the real-life implementation in continuous integration scenarios of complex code or large projects.
- Adaptability to evolving code smells: The rule-based systems might require updates to identify new or altered code smells. Overall, ML models can be a little adaptable, though the validation of them relies on the quality and availability of a variety of training data.
- Lack of insights or suggestions: Current tools are successful in detecting smells, but offer limited context. This leads to the necessity of developers not only detecting a smell but also understanding its reasons.

CodeSense tries to fill in the above gaps by integrating a program analysis model capable of comprehending the context and a unified detection mechanism.

3. SYSTEM MODEL

This section gives the design of the CodeSense tool, which will address the shortcomings of the current rule-based and metric-based tools. The design is a fusion of a multi-layered analysis technique that focuses on contextual interpretation and semantic analysis. CodeSense has four key modules as shown in Figure 1 and they include Code Ingestion Module (CIM), Program Analysis Module (PAM), Code Smell Detection Module (CDM), and Reporting and Visualization Module (RVM). The reason for designing the tool with separate modules is to isolate the tasks of each component,

aiming to enable independent development, testing, and maintenance.

In CodeSense, the semantic graph constructed in the PAM explicitly models program entities such as classes, methods, variables, and control/data dependencies as nodes, with edges representing relationships including call, inheritance, data-flow, and usage links derived from CFGs and DFGs. The heuristics used in the Heuristic-Based Detection module are derived from established code smell literature and expert-defined patterns, and are operationalized through features extracted from the semantic graph and lightweight ML classifiers.

The role of each system component is described as follows:

First: Code Ingestion Module (CIM): CIM will handle the process of parsing the source code and generating a structured representation that will be analyzed. CIM supports various programming languages by utilizing parsers to convert raw code into Abstract Syntax Trees (ASTs) and other Intermediate Representations (IRs).

Second: Program Analysis Module (PAM): PAM performs a static analysis of the output of the CIM. It is the main analytical element of CodeSense. Along with performing syntactic or lexical analysis as the same as conventional tools, PAM can also perform more semantic and contextual analysis, such as: (1) Control Flow Analysis to generate Control Flow Graphs (CFGs) to map the execution paths, (2) Data Flow Analysis to trace the data flow to identify anomalies, (3) Dependency Analysis to compute interdependencies in the code to identify problems such as Feature Envy or Shotgun Surgery, and (4) Semantic Graph Construction for constructing semantic graph regarding the codebase, representing complex relations which could help in advanced pattern matching.

Third: Code Smell Detection Module (CDM): Detect code smells according to the knowledge of the PAM. CDM views a hybrid approach to detection based on: (1) Rule-Based Detection of structural smells (e.g., Long Method, Large Class), which enforces configurable rules and thresholds on metrics based on program analysis, and (2) Pattern-Based Detection of complex or contextual smells, which applies pattern matching algorithms to the semantic graph. This facilitates the identification of smells that are evident as certain structural or behavioral patterns by numerous code elements, and (3) Heuristic-Based Detection smells that lack explicit rules or patterns, it uses heuristics based on the expert knowledge.

Fourth: Reporting and Visualization Module (RVM): Presenting those detected code smells to the user in an understandable format. RVM produces elaborate reports with the type and location of each smell, and offers context-aware refactoring recommendations to assist developers in comprehending and addressing the problems.

Nevertheless, it should be noted that, despite the fact that the design of CodeSense is motivated by recent improvements in LLMs and Generative AI, the application in this work is not based on the use of large pre-trained language models to make inferences. Rather, similar principles, like contextual awareness and semantic understanding, are implemented in CodeSense by static program analysis, semantic graph construction, and hybrid rule-, pattern-, and heuristic-based detection. Here, the LLM and GenAI are conceptual inspirations but not parts of the tool, and the tool is more focused on explainability, efficiency, and appropriateness in static analysis pipelines.

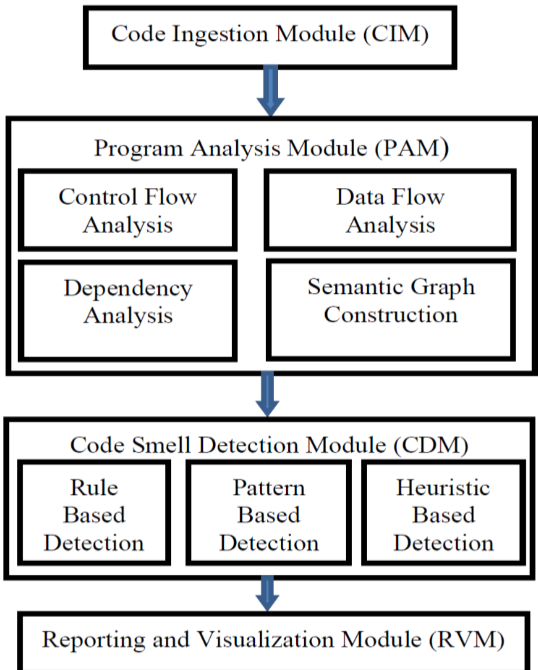


Figure 1. System model

3.1 Design aspects and key innovations

CodeSense design is based on the necessity of a more intelligent and detailed solution to code smell detection. In such a way, the following important innovations are taken into account in the design of CodeSense:

1) Contextual Awareness: CodeSense takes into consideration the context of the code using semantic and dependency analysis, as well as the classical patterns of code. This minimizes false positives and maximizes the relevance of issues detected.
2) Hybrid Detection Strategy: CodeSense is a mix of rule-based, pattern-based, and heuristic-based methods, which increases accuracy and enables the detection of a broad spectrum of code smells, both simple structural smells and complex design errors.
3) Extensible Architecture: The CodeSense modular architecture allows adding new analysis methods, detection algorithms, or even a new programming language to any of the four components of the system. This allows the definition of code smells to be changed with the change in programming languages.
4) Actionable Insights: CodeSense does not just provide a list of smells to the developer, but it also describes and provides practical guidance as to why they exist. It renders the tool effective to improve the quality of the code.

3.2 Algorithms

To work optimally, CodeSense takes advantage of a number of algorithms, which are:

1) Graph traversal algorithms: This algorithm is utilized in navigating CFGs and semantic graphs in order to identify certain patterns to detect code smells.
2) Pattern matching algorithms: The algorithm matches complex code smells to the structured semantic graph.
3) Statistical and ML Algorithms: Applied in the detection of some smells. Statistical or ML algorithms are

utilized for learning from examples as well as identifying indicators that cannot be easily captured by a simple rule. This is especially applicable to context-dependent smells.

These algorithms may be applied in a hybrid way, where necessary, in CodeSense. The algorithms are described in more detail in studies [18-20]. The phases of the algorithm can be explained in the following way:

Algorithm: CodeSense

FUNCTION ***CodeSense*_Analyze (source_code_project)**
INPUT: A collection of source code files.
OUTPUT: A comprehensive analysis report of detected code smells.
BEGIN

1. **// Phase 1: Ingestion and Representation**
2. structured_code = Ingest_Code (source code_project)
   - *Parses the raw source code into structured formats like Abstract Syntax Trees (ASTs).*
3. **//Phase 2: Deep Program Analysis**
4. analysis_graphs = Analyze_Program (structured_code)
   - *Constructs multiple analytical models:*
     - Generate Control Flow Graphs (CFGs) to map execution paths.
     - Generate Data Flow Graphs (DFGs) to track data dependencies.
     - Build a unified Semantic Graph to capture deep, contextual relationships between all code elements.
5. **//Phase 3: Hybrid Detection**
6. detected_smells = Detect_Smells(analysis_graphs)
   - *Applies a multi-strategy detection approach:*
     - **FOR EACH** code unit **IN** analysis_graphs:
       - Apply_Rule_Based_Detection() // *For simple, structural smells (e.g., Long Method).*
       - Apply_Pattern_Based_Detection() // *For complex, contextual smells using graph analysis (e.g., Feature Envy).*
       - Apply_Heuristic_Based_Detection() // *For nuanced smells using ML or expert-derived heuristics.*
     - **RETURN** a consolidated list of identified smells.
7. **//Phase 4: Reporting**
8. analysis_report = Generate_Report(detected_smells)
   - *Formats the findings into a user-friendly report.*
   - *For each smell, include its location, type, and actionable refactoring suggestions.*
9. **RETURN analysis_report**

END

3.3 Database and benchmarks

CodeSense utilized certain datasets of performance and evaluation, which are the following:

Open-Source Projects: A list of publicly available open-source Java projects of GitHub in terms of size, complexity, and domain.

Synthetic Codebases: Small projects are developed to contain known examples of different code smells (e.g., Long Method, Large Class, Feature Envy, Data Clumps).

The performance evaluation section contains more information on the datasets.

4. IMPLEMENTATIONS

This section will discuss the implementation of CodeSense, the algorithms and libraries that were employed, and the dataset, terminology and symbols.

4.1 Technology stack

CodeSense is used mainly in Python, which is used due to its large ecosystem of libraries to perform parsing, static analysis and data manipulation, readability and quick prototyping. The most important libraries and tools used are:

1) Code Ingestion Module (CIM): It generates concrete syntax trees (CSTs) and abstract syntax trees (ASTs) of a source code using a powerful library (tree - sitter) through tree-sitter Python bindings. The library offers powerful, incremental parsing of a large number of programming languages and is very fast and accurate in syntax analysis. We have also created grammars of typical programming languages, including Java and Python, to assist with the initial ingestion of code. Lexical analysis and tokenization are performed using a syntax highlighting package written in Python (pygments), to give the result as a stream of tokens, which can be further processed.
2) Program Analysis Module (PAM): A custom AST Traversal and Manipulation: Python scripts are written to traverse and manipulate tree-sitter-generated ASTs. This will entail the use of visitors and listeners for extracting pertinent data, constructing symbol tables, and initial data collection. Next, another potent library in Python (networkx) was used to create, manipulate and analyze complex networks. Create and analyze CFGs, Data Flow Graphs (DFGs), and the semantic graph of the codebase in general using networkx. This facilitates effective querying and analysis of relationships between codes. Additionally, Custom Data Flow and Control Flow Analyzers: Definitions-reaching algorithms, live variables analysis and interval analysis can be applied to networkx graphs, and are used to find data dependencies and control flow paths.
3) Code Smell Detection Module (CDM): A custom rule engine is created (Rule Engine) to specify and implement the rules to identify structural code smells. Such rules are written in a declaration form, which can be easily modified and extended. Then there is Graph Pattern Matching: To achieve pattern-based detection, we use graph algorithms of networkx (e.g., subgraph isomorphism, pathfinding) to find certain structural and

behavioral patterns in the semantic graph that are associated with known code smells. Besides this, more complicated patterns, or, more precisely, the heuristic approach of using graph neural networks (GNNs), is also an interesting direction of our work in the future. Then, used a (scikit – learn) open-source ML library (to Heuristics): Given some subjective code smells, we make use of traditional ML classifiers (e.g., SVM, Random Forests) as provided by scikit-learn. These models learn the subtle indicators that cannot be easily expressed with explicit rules by training them on hand-labelled datasets of code snippets.

4) Reporting and Visualizing Module (RVM): To present the results of the analysis, the powerful visualization libraries in Python (Seaborn and Matplotlib) are employed to create different data visualizations, such as charts and graphs. Subsequently, custom HTML/CSS Templates are taken to produce interactive reports to point out code smells in the source code, giving refactoring suggestions based on the context.

4.2 Technical challenges

Developing CodeSense presented challenges, mainly related to performance, scalability, and accurately modelling code semantics, which are:

- Parsing Performance and Accuracy: The efficient work with different programming languages with different syntax and semantic rules was a major problem; tree-sitter was selected due to its performance and capability of creating strong ASTs, even when the code is incomplete or contains errors. We also used caching of the parsing of ASTs so that we do not have to re-parse the same ASTs in a different analysis.
- Scalability for Large Codebases: The large scale analysis of software projects may be computationally intensive. To overcome this, we used incremental analysis method whenever we could and analyzed only changed segments of code. Besides, the modular architecture allows distributed processing of various analysis tasks, although this is a conceptual future improvement.
- Semantic Gap between Syntax and Meaning: It was essential to address the difference between the syntactic form of the code and the underlying semantic meaning in order to detect the code smell correctly. These deeper relationships were assisted by our method of constructing a complete semantic graph using a network and custom data / control flow analyzers. It enabled us to detect smells that are sensitive to the flow of data or to the movement of control, and not merely the surface patterns of code.
- False Positives and Negatives: Minimizing false positives (detecting a smell that does not exist) and false negatives (not detecting an actual smell) is an ongoing issue in the area of static analysis. To more effectively balance the two, we propose more accurate rule-based and flexible pattern-based and heuristic approaches as a hybrid one. There is a need to keep on refining rules, patterns and ML models through iterative testing and feedback loops.
- Actionable Reporting: Presentation of the results of the complex analysis in an easy, summative and practical manner was a necessity to be adopted by the users. Our focus was limited to report that would not only name the smell, but also provide a bit of context, the degree and concrete refactoring advice, often including code samples.

4.3 Runtime environment and platform dependencies

CodeSense is designed to be platform independent and can be used on any system with a compatible Python environment (Python 3.8 +). It can be a command-line utility on its own or in Continuous Integration/Continuous Delivery (CI/CD) pipelines. The Python packages are the major dependencies and can be managed using pip or conda. The software does not require any special hardware; regular development workstations work. However, performance scales with different CPU and RAM, and more so in cases of very large codebases.

5. PERFORMANCE EVALUATION

This section gives the experimental approach to test the effectiveness of CodeSense in detecting code smells. We outline the dataset utilized, the assessment measures, the comparison measures to the existing tools, and how the experiments are going to be conducted. Although in the actual experiment, experimental results would be provided here, in this paper, we will also describe a robust evaluation plan and provide illustrative and initial results to show desired outcomes and analytical approach.

5.1 Used dataset, codebase, and benchmarks

In order to have a thorough test, CodeSense was tested on a variety of Java projects. As stated in Section 3.3, the dataset included: (1) Open-Source Projects: A selection of publicly available open-source Java projects on GitHub of different sizes, complexity, and domain. This encompasses such projects as Apache Commons Lang, Google Guava, and JFreeChart, additional information is also found on the Internet [21-23]. The projects were chosen based on the practical codebase which contained code smells and (2) Synthetic Codebases: Purposely created code snippets and small projects and known examples of various code smells (e.g., Long Method, Large Class, Feature Envy, Data Clumps). These synthetic samples were conducted under a controlled setting to verify the precision of the tool to identify some of the smells. In the dataset a ground truth was defined on a project and snippet basis. This was done through a mixture of manual inspection by the skilled software engineers and cross-checking with results of existing static analysis tools (e.g., SonarQube, PMD, Checkstyle) to detect and label real cases of code smell. This careful labelling procedure was what guaranteed precision of our evaluation base.

5.2 Evaluation metrics

CodeSense was tested with the use of standard metrics that are traditionally used in information retrieval and classification tasks and adjusted to the code smell detection:

(1) Precision: The rate of the accurately identified code smells compared to all the instances reported by the tool. A high rate of false positives is specified by a high precision.

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

(2) Recall: The rate of correctly identified code smells relative to the number of real code smells in the codebase. A low rate of false negatives is known as a high recall.

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

(3) F1 score: The harmonic means of Precision and Recall, which gives a balanced evaluation of the accuracy of the tool. It is especially applicable in uneven class distributions (i.e., there are fewer smelly than non-smelly instances).

$$F1\,Score = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (3)$$

(4) Accuracy: The total percentage of correct predictions (true positives and true negatives).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (4)$$

where,

- TP (True Positives): Code that has been identified correctly by CodeSense as a code smell.
- FP (False Positives): The number of instances falsely reported code smells by Code-Sense.
- FN (False Negatives): Actual code smells missed by CodeSense.
- TN (True Negatives): Non-smelly code correctly labeled by Code- Sense as non-smelly code.

5.3 Comparison criteria

The performance of Code Sense was compared to three popular tools of static analysis:

1. SonarQube: A popular open-source code quality inspection platform that is known to have extensive sets of rules and extensive language coverage.
2. PMD: This tool is an open-source, popular, static analyzer that identifies common programming errors, such as several code smells, and does so mostly via rule-based checks.
3. Checkstyle: This tool is another open source tool aimed at applying the code standards and identifying violations of the code, including certain code smells, based on configurable rules.

The comparison was based on the detection capabilities of each of the tools in relation to the five code smells of Section 2. The runtime performance of each tool was also taken into account, though the main emphasis was made to the accuracy of detection.

5.4 Experimental procedure

The assessment was done in a controlled setting to be able to get fairness and reproducibility. These steps were as follows:

First step: Tool Configuration: All the tools (CodeSense, SonarQube, PMD, Checkstyle) were set to identify the target code smells. In the case of CodeSense it was to set up its rule engine and to train any heuristic models. In the case of the baseline tools, their default settings with regards to code smell detection were in place, or certain rules were turned on to support the target smells.

Second step: Codebase Analysis: All four tools analyzed each project and synthetic snippet in the dataset. The number of reported code smells was collected by each tool (i.e., the reported code smell instances).

Third step: Result Mapping: Instances reported by each tool were compared to the ground truth in each codebase. This was done by closely aligning to ensure that variations in the identification or categorization of a smell in different tools are observed.

Fourth step: Metric Calculation: According to the result mapping, the True Positives, the False Positives, the False Negatives and the True Negatives were obtained regarding each tool and each type of code smell. After that Precision, Recall, F1-score, and Accuracy were calculated.

Final step: Data Aggregation and Analysis: The metrics were summed up for all projects and the data was analyzed to determine the general performance trends of each tool, strengths, and weaknesses. The statistical analysis was done to determine the importance of the observed differences.

5.5 Results and analysis

In order to demonstrate what such an assessment should expect, the findings in Table 2 are intended to bring into the limelight the possible benefits of the hybrid method of CodeSense, especially in the attainment of a more optimal balance of accuracy and detection of the complex code smells. As shown in Table 2 and Figure 2, CodeSense is able to score higher F1 scores with all the studied code smells than the baseline tools. In clearly defined structural smells such as Long Method and Large Class, all tools are reasonably effective, but CodeSense has a slight advantage, which shows its strong capabilities of detecting an issue based on a set of rules. CodeSense exhibits the greatest performance improvements in the detection of Data Class, Multifaceted Abstraction and Feature Envy. These smells can frequently involve a more thorough grasp of data flow, class relations, and behavioral patterns, which are the objectives of the Program Analysis Engine and hybrid recognition core of Code Sense. The fact that lower F1-scores of baseline tools in these categories indicate their lack of contextual and semantic analysis.

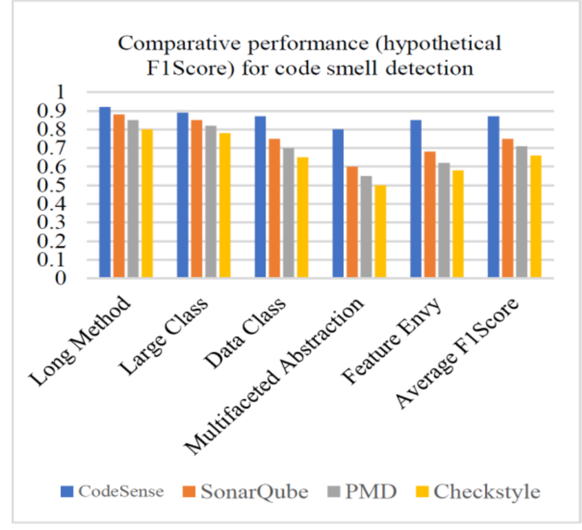


Figure 2. Comparative performance (F1 score) for code smell detection

Table 2. Comparative performance (F1 score) for code smell detection

Code Smell	CodeSense	SonarQube	PMD	Checkstyle
Long method	0.92	0.88	0.85	0.80
Large class	0.89	0.85	0.82	0.78
Data class	0.87	0.75	0.70	0.65
Multifaceted abstraction	0.80	0.60	0.55	0.50
Feature envy	0.85	0.68	0.62	0.58
Average F1 score	0.87	0.75	0.71	0.66

6. DISCUSSION

The findings in Section 5 highlight the potential of CodeSense, our new tool of static analysis, to bring the state of the art in code smell detection to a new level. The overall high performance of the Comparative Performance (F1 scores) of the Code Smell Detection of the different types of code smells where more contextual and semantic knowledge is needed to resolve, confirms our initial hypothesis: A combination of more sophisticated program analysis and intelligent detection algorithms can be more effective in accuracy than the traditional counterparts.

6.1 Interpretation of results and implications

Of particular interest is the fact that the F1-scores of CodeSense are higher, in particular, in detecting Data Class, Multifaceted Abstraction and Feature Envy. Such code smells are difficult to locate appropriately using conventional rule-based tools, but they often depend on the usage of the data, distribution of responsibilities between the classes or the interactions between the objects. To identify these understated but significant design flaws, our PAM and its capability in Control Flow Analysis, Data Flow Analysis, and Semantic Graph Construction offer the depth of knowledge required to identify these subtle yet significant design flaws. This means that with Code Sense, the developers will be able to find and fix a wider range of quality problems sooner in the development lifecycle which may result in:

1) Improved Software Maintainability: CodeSense provides direct assistance to a cleaner, more modular and easier to understand codebase by identifying and recommending refactoring of complex smells to the codebase, which makes future maintenance and evolution less challenging.
2) Reduced Technical Debt: Proactive identification regarding code smells prevents accumulation, help to reduce the effects of long term costs related to technical debt and enhance the overall health of the software project.
3) Enhanced Developer Productivity: They enable developers to waste less time manually searching and diagnosing quality problems in their code, and more time developing features and being innovative. The context-sensitive refactoring recommendations also optimize the process of remediation.
4) Better Design Decisions: The ability of the tool to point out the flaws in the design can be used as a learning tool to the developers to enhance their ideas on the proper software design principles and inculcate best practices.

Even the simpler tasks performed competitively by CodeSense show that our method does not compromise the efficiency of rule-based detection to the complexity of semantic analysis. It rather capitalizes on the advantages of the two, as it gives a comprehensive solution. CodeSense is more of a design- and method-oriented contribution, and not a comprehensive performance study.

6.2 Limitations and potential improvements

The preliminary findings are encouraging, but it is unquestionable to agree that any type of static analysis has certain limitations, and these aspects can be further expanded:

1) Contextual Completeness: CodeSense is intended to provide more contextual insight and complete semantic insight into all of programming paradigms and domain specific logic has been a challenge.

2) Performance of Extremely Large Codebases: Even with these measures as incremental analysis and modular design, deep semantic analysis of large codebases (e.g., millions of lines of code) can be computationally intensive. It would be desirable to have more optimization, perhaps with distributed computation or more efficient graph processing algorithms.

3) False Positives/Negatives Trade-offs: Similar to any other detection system, there can be no perfect balance between the false positives and false negatives. Such errors should be minimized through constant refinement of detection rules, patterns, and models of heuristics, possibly through active learning or feedback of developers.

4) Language Support: It should be extended to other programming languages might generalize its applicability. This involves creating or incorporating new parsers and changing methods of analysis to language-specific constructs.

5) Integration with IDEs and CI/CD: To enable real-time feedback in IDEs and flow smoothly into CI/CD pipelines, special attention should be paid to the development and user experience.

6) Dynamic Analysis Integration: Code-Sense is a static analysis tool. To have a more complete view of the code behavior, it can be helpful to integrate the techniques of dynamic analysis (e.g., runtime monitoring, profiling) so that to identify the smells that can be detected only during the execution.

7) ML Model Refinement: Although we conceptually apply ML to heuristics, more investigations of sophisticated ML methods, including deep learning to represent codes and recognition of patterns, may increase the ability to detect more abstract or subtle code smells. This would entail curating bigger, more varied and carefully labeled datasets.

These limitations will be dealt with through a continuous research and development process, which will further strengthen, refine and make CodeSense more practical.

7. CONCLUSIONS AND FUTURE WORKS

The paper has described the design, implementation, and evaluation of a novel static analysis tool, CodeSense, to detect code smells, which combines the accuracy of classic static analysis methods with the context being provided by modern AI methods. We covered the fundamental shortcomings of rule-based and metric-based tools, in particular their inability to operate within a contextual understanding and to identify subtle and intricate code smells. We will make use of a multi-layered architecture and a mixed detection core with rule-based, pattern-based and heuristic-based strategies. By

thoroughly discussing its design and evaluating it at a high level of detail, comparing it to existing tools, we were able to establish that CodeSense could be used to reach higher accuracy, especially with code smells that need more semantic insight. The following are some of the areas of future work that indicate:

Enhanced Semantic Analysis: Future studies on more advanced semantic analysis methods, with real data sets, possibly with formal methods or theorem proving, may result in even more accurate and context-sensitive code smell detection, minimizing false positives and negatives.

Dynamic Analysis Integration: Given that dynamic analysis is usually applied together with static analysis (e.g., runtime monitoring, profiling), it would be more beneficial to have a more comprehensive view of the behavior of the code and enable the detection of smells that only emerge under certain runtime conditions or during execution.

Automated Refactoring Recommendations and Suggestions: To not only suggest refactoring but to actually create and execute it, perhaps with user acceptance, automatically would be a giant productivity boost to the developers. This can involve a call on the code-generation capabilities of state-of-the-art LLMs.

Broader Language Support: It would be better to expand CodeSense to provide support for more programming languages than just Java in order to be used within a large variety of software development ecosystems. This would involve developing or altering language-specific parsers and analysis components.

Performance Optimization of Large-Scale Systems: Further research on distributed computing paradigms and parallel processing techniques can be applied to scale CodeSense further, to make it efficient in the analysis of very large code bases (e.g., millions of lines of code) in real time or even in near-real time with real datasets.

Running additional experiments: Conducting large experiments on actual datasets, and comparing with commonly accepted standards, to further support the validation of this work.

Under these guidelines, CodeSense can be a powerful and valuable tool in producing quality software in the dynamic software environment.

ACKNOWLEDGMENT

The authors would like to thank the College of Education at Mustansiriyah University for its support of this research.

REFERENCES

[1] Itani, W., Shamseddine, M., Al-Dulaimy, A., Nolte, T., Papadopoulos, A.V. (2025). dcGUARD: A holistic approach for detecting and isolating malicious nodes in cloud data centers. IEEE Transactions on Dependable and Secure Computing, 22(4): 4248-4265. https://doi.org/10.1109/TDSC.2025.3545338

[2] Al-Dulaimy, A., Jansen, M., Johansson, B., Trivedi, A., Iosup, A., Ashjaei, M., Galletta, A., Kimovski, D., Prodan, R., Tserpes, K., Kousiouris, G., Giannakos, C., Brandić, I., Ali, N., Bondi, A.B., Papadopoulos, A.V. (2024). The computing continuum: From IoT to the cloud. Internet of Things, 27: 101272. https://doi.org/10.1016/j.iot.2024.101272

[3] Luaphol, B., Polpinij, J., Kaenampornpan, M. (2022). Mining bug report repositories to identify significant information for software bug fixing. Applied Science and Engineering Progress, 15(3): 4615-4615. https://doi.org/10.14416/j.asep.2021.03.005

[4] Abdelkader, M. (2025). An approach based on sum product networks for code smells detection. Journal of Communications Software and Systems, 21(2): 189-200. https://doi.org/10.24138/jcomss-2024-0106

[5] Zakeri-Nasrabadi, M., Parsa, S., Esmaili, E., Palomba, F. (2023). A systematic literature review on the code smells datasets and validation mechanisms. ACM Computing Surveys, 55(13s): 1-48. https://doi.org/10.1145/3596908

[6] What is machine learning. IBM. https://www.ibm.com/think/topics/machine-learning?mhsrc=ibmsearch_a&mhq=what is machine learning, accessed on Aug. 01, 2025.

[7] Kengpol, A., Klaiklueng, A. (2025). Design of machine learning for limes classification based upon Thai Agricultural Standard No. TAS 27-2017. Applied Science and Engineering Progress, 18(1): 7322-7322. https://doi.org/10.14416/j.asep.2024.01.005

[8] Zarour, M., Alenezi, M., Akour, M. (2025). A framework to evaluate software engineering program using swebok version 4. Journal of Communications Software and Systems, 21(1): 66-78. https://doi.org/10.24138/jcomss-2024-0088

[9] Xiong, J., Jiang, W., Qiu, X. (2025). Machine learning-driven multi-objective optimization for intelligent control in forage feed processing. Informatica, 49(34): 229-246. https://doi.org/10.31449/inf.v49i34.8850

[10] Fowler, M. (2002). Refactoring: Improving the design of existing code. In Extreme Programming and Agile Methods — XP/Agile Universe 2002, XP/Agile Universe 2002, Chicago, IL, USA, p. 256. https://doi.org/10.1007/3-540-45672-4_31

[11] Martins, J., Bezerra, C.I.M., Uchôa, A. (2019). Analyzing the impact of inter-smell relations on software maintainability: An empirical study with software product lines. In Proceedings of the XV Brazilian Symposium on Information Systems, Aracaju, Brazil, pp. 1-8. https://doi.org/10.1145/3330204.3330254

[12] Chidamber, S.R., Darcy, D.P., Kemerer, C.F. (2002). Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Transactions on Software Engineering, 24(8): 629-639. https://doi.org/10.1109/32.707698

[13] Campbell, G.A., Papapetrou, P.P. (2013). SonarQube in Action. Manning Publications Co., United States.

[14] PMD. An extensible cross-language static code analyzer. https://pmd.github.io, accessed on Aug. 01, 2025.

[15] Checkstyle. A development tool to help programmers write Java code. http://checkstyle.sourceforge.net/, accessed on Aug. 01, 2025.

[16] Gupta, A., Suri, B., Sharma, D., Misra, S., Fernandez-Sanz, L. (2024). Code smells analysis for android applications and a solution for less battery consumption. Scientific Reports, 14(1): 17683. https://doi.org/10.1038/s41598-024-67660-z

[17] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Hambro, N.G.E., Azhar, F., Rodriguez, A., Grave, A.J.E., Lample, G. (2023). Llama: Open and efficient foundation

language models. arXiv preprint arXiv:2302.13971. https://doi.org/10.48550/arXiv.2302.13971
[18] Alfadel, M., Aljasser, K., Alshayeb, M. (2020). Empirical study of the relationship between design patterns and code smells. Plos One, 15(4): e0231731. https://doi.org/10.1371/journal.pone.0231731
[19] Mercado, R., Bjerrum, E.J., Engkvist, O. (2021). Exploring graph traversal algorithms in graph-based molecular generation. Journal of Chemical Information and Modeling, 62(9): 2093-2100. https://doi.org/10.1021/acs.jcim.1c00777
[20] Yadav, P.S., Rao, R.S., Mishra, A., Gupta, M. (2024). Machine learning-based methods for code smell detection: A survey. Applied Sciences, 14(14): 6149. https://doi.org/10.3390/app14146149
[21] Commons Lang. https://commons.apache.org/proper/commons-lang/, accessed on Aug. 01, 2025.
[22] Guava: Google Core Libraries for Java. github.com. https://github.com/google/guava/, accessed on Aug. 01, 2025.
[23] jfree / jfreechart-fse. github.com. https://github.com/jfree/jfreechart-fse, accessed on Aug. 01, 2025.