A Multi-Approach Evaluation of Python Code Smell Detection Using Heuristics, Learning Models, and Language Models

Igor Soares de Oliveira
PUC-Rio
Rio de Janeiro, Brazil
isoliveira@inf.puc-rio.br

Alexandre Andrade
PUC-Rio
Rio de Janeiro, Brazil
aandrade@inf.puc-rio.br

Saymon Souza
UFMG
Minas Gerais, Brazil
silvasouzasaymon@dcc.ufmg.br

Eduardo Figueiredo
UFMG
Minas Gerais, Brazil
figueiredo@dcc.ufmg.br

Juliana Alves Pereira
PUC-Rio
Rio de Janeiro, Brazil
juliana@inf.puc-rio.br

Abstract

Code quality is fundamental to software maintainability and evolution, making code smell detection an important task in automated quality assessment. This paper presents a comparative evaluation of automated approaches for Python code smell detection across six smell types: Long Method (LoM), Long Parameter List (LPL), Large Class (LC), Lazy Class (LzC), Data Class (DC), and Magic Number (MN). We compare AST-based heuristics, Machine Learning (ML), Deep Learning (DL), and Language Models (LMs), while also investigating the impact of prompt engineering strategies, including zero-shot, one-shot, and Chain-of-Thought (CoT) prompting with and without structural metrics. To support this evaluation, we introduce `PyHub-SMELL`, a manually annotated dataset of Python code fragments. Our results show that prompt engineering significantly affects LM performance, particularly for semantically complex smells, whereas simpler prompting strategies remain effective for syntactically explicit ones. Overall, learning-based approaches consistently outperform heuristic-based detection, with ML providing the most stable performance and LMs demonstrating strong potential for semantically richer code smell detection.

Keywords

Software Quality, Code Smells, Python, Large Language Models, Learning Models

1 Introduction

High code quality is essential for software development [33]. Fowler et al. [15] define code smells as indicators of structural problems in code design that may negatively affect the maintenance and evolution of a software system. According to Martin [32], the cost of maintaining a poorly structured and messy codebase increases significantly over time, leading to reduced developer productivity and hindering the system's evolution. Empirical evidence indicates that substandard programming practices can lead to a 15-fold increase in functional defects and significantly hinder developers' ability to understand program logic and support effective code comprehension [49]. These challenges highlight the need for effective techniques to identify and mitigate structural issues in codebases at early stages. In this context, code smell detection has emerged as a key strategy for supporting developers in maintaining code quality and preventing the accumulation of technical debt. Among the various types of code smells discussed in the literature, common examples include Long Method (LoM), Long Parameter List (LPL), Large Class (LC), Lazy Class (LaZ), Data Class (DC), and Magic Number (MN) [15, 38]. These smells capture recurring structural and implementation issues, such as excessive method complexity, over-parameterization, poorly structured or underutilized classes, and the use of unexplained numeric literals, all of which can negatively impact code comprehension and maintainability.

Recent research has increasingly focused on the transition from traditional heuristic-based detection to flexible learning-based models to improve code smell detection [2, 5, 7, 20, 28, 35]. Early efforts predominantly used Machine Learning (ML) techniques, such as Random Forest and Decision Trees, which leverage software project metrics to identify structural patterns associated with design flaws [2, 20]. To capture more complex, non-linear relationships within large-scale datasets, previous studies have successfully applied Deep Learning (DL) techniques [5, 28]. More recently, the emergence of Language Models (LMs) has introduced semantic-driven code smell detection [42, 46, 55, 59]. Preliminary investigations have demonstrated that performance can be significantly improved through strategic prompt engineering, although challenges remain in the handling of highly complex or subjective patterns [46]. Collectively, these diverse approaches reflect a growing trend towards using automated reasoning to overcome the rigidity of traditional heuristic-based detection tools [23].

In contrast to existing studies that focus on a single detection approach, our work provides a comprehensive, multi-approach evaluation within a unified framework. We introduce a unified framework for code smell detection, which includes a diverse set of smell types and multiple detection approaches, including heuristics based on Abstract Syntax Trees (ASTs), ML, DL, Small Language Models (SLMs) and Large Language Models (LLMs). By evaluating these diverse approaches using a human-validated dataset of real-world Python code fragments, our work addresses unique challenges that are frequently overlooked in the literature. Additionally, LMs introduce a layer of interpretability by generating natural-language justifications for identified smells, providing a qualitative depth that traditional learning-based models lack.

This work replicates a previous study [1], which provided an initial investigation into the use of SLMs for code smell detection in Python software systems. While the prior study offered valuable preliminary insights, it was limited in scope, focusing on a restricted

set of code smells, SLMs, and a narrow range of prompting strategies. Additionally, it did not provide an integrated tool to support practical adoption. In contrast, our study significantly broadens this investigation. We incorporate a wider range of prompt engineering strategies and include both SLMs and LLMs. Additionally, we introduce a manually annotated dataset to support an in-depth analytical evaluation of model performance across diverse configurations.

To achieve these goals, we propose a multi-phase experimental design aimed at systematically evaluating the effectiveness of LMs in identifying and classifying code smells in Python code. Our contributions are the following: (1) We implemented `Scylla`[1], a unified and extensible framework that integrates heuristic-based rules, ML, DL and LMs into a single workflow. This design enables scalable and reproducible code smell detection across multiple approaches. (2) We constructed a dataset, named `PyHub-SMELL`[2], through a rigorous human-driven, double-validation annotation process. In this process, Python developers labeled code fragments to establish high-quality ground truth for training and evaluating detection models. (3) We integrated LMs using the Ollama framework [37], benchmarking a diverse set of architectures comprising five families of models and eight LMs, evaluated across six prompting strategies. (4) We performed a comparative analysis of learning-based approaches, including 15 traditional ML algorithms, 4 DL architectures, and 8 LMs. This evaluation investigates whether these approaches can achieve predictive performance comparable to, or exceeding, that of a traditional heuristic-based approach, while also considering trade-offs in interpretability and practical applicability.

The results of this study provide a comprehensive assessment of automated approaches for Python code smell detection. First, prompt engineering was found to significantly impact the performance of LMs, with reasoning-based strategies such as Chain-of-Thought improving results for semantically [3] complex smells, achieving F1-scores up to 0.84 for LC and 1.00 for LzC, while simpler zero-shot strategies remained competitive for syntactically explicit smells such as LPL and MN. Second, in the cross-approach comparison, learning-based approaches consistently outperformed the heuristic-based baseline, with traditional ML models achieving the most stable and highest overall performance, reaching F1-scores up to 0.91, whereas LMs proved competitive for semantically richer smells, in some cases matching or surpassing ML baselines. Overall, the findings indicate that code smell detection effectiveness depends on the interplay between model approach, prompt design, and smell type.

2 Background and Related Work

Code smells are symptoms or indicators of technical debt that hinder long-term maintenance and seamless evolution [15]. Given their impact on development overhead and system reliability, accurate identification and prioritization for refactoring remain critical research challenges [13, 14, 47, 51, 52]. Refactoring involves modifying software architecture to remove code smells and improve maintainability while preserving external behavior [15].

Extensive research has explored approaches for code smell detection, ranging from manual audits [31, 50] to automated heuristic-based tools [13, 47, 51, 52]. Despite their relevance, these tools are often limited by rigid metric-based rules and complex setup requirements [2, 31]. Tools, such as *PySmell* [7], *Dpy* [5], and *PyExamine* [45], present limitations. They are either discontinued, lack active maintenance, are not open source, or lack integration of different approaches in a single environment, which restricts their extensibility and reproducibility in contemporary research and industrial contexts. Moreover, empirical studies report divergence between tool outputs and expert judgment, hindering adoption in practice [2, 6, 10, 20, 35, 57].

Recent work has shifted toward ML [2, 20, 35] and DL [3, 5, 7, 25, 28, 56] approaches. ML techniques perform well for structurally defined smells, such as Large Class and Data Class [2, 20], while DL is more effective in capturing complex, non-linear relationships [28, 35]. Despite these advances, most studies rely on heuristic-based tools for ground truth, introducing bias and reducing reliability [6, 13, 30]. The scarcity of manually labeled datasets continues to hinder progress [2, 28]. Additionally, most studies are not for Python, and model generalization across projects and languages remains limited. Recent studies have explored LMs to overcome limitations of traditional approaches [42, 46, 55, 59]. Although LMs are agnostic to programming language, there are still missing datasets in the literature to evaluate their performance for Python. Early datasets [7] lack traceability, while more recent ones [1] are restricted to a very small number of smells. To address this gap, we introduce `PyHub-SMELL`, a large-scale dataset with 32,392 labeled instances, spanning multiple detection perspectives and enabling reproducible and comprehensive evaluation across six code smell types.

Moreover, our work differs from related works by providing a broader empirical comparison of SLMs and LLMs within a unified framework. We propose `Scylla`, an open-source multi-approach tool for Python code smell detection that leverages heuristics, ML, DL, and LMs in a single workflow. This integration enables cross-project analysis and supports qualitative assessment of different approaches. In contrast to existing tools, `Scylla` integrates multiple detection approaches, enabling deeper analysis and improved interpretability based on agreement and disagreement across approaches. Its distributed architecture promotes scalability, modularity, and long-term evolution.

3 Study Design

This section presents our study design, structured into four phases: (1) code extraction and data processing, (2) labeling process, (3) Scylla, and (4) data analysis.

3.1 Code Extraction and Data Processing

In this phase, we employed a tool called *GHS* [9]. GHS identifies and retrieves GitHub repositories based on configurable characteristics, such as programming language, number of commits, number of contributors, and other repository-level filters. For the purposes of

[1] `Scylla` takes its name from a famous creature in Greek mythology. Like the mythical guardian lurking in treacherous waters, `Scylla` symbolizes vigilance against hidden dangers (*i.e.* code smells) that can hinder code comprehension. The name reflects the tool's purpose to detect and expose such threats before they "devour" code quality.

[2] A Central Hub of Human-Annotated Python Code Smells

[3] Semantic interpretation refers to reasoning about the responsibilities and organization of code elements beyond explicit metric thresholds, rather than relying solely on quantitative measures derived from source code

this study, we configured the search to retrieve repositories whose primary programming language was Python with at least ≥ *57,300* stars, which we adopted as a proxy for project popularity, maturity, and sustained community engagement.

The filtering process resulted in a corpus of *17* popular open-source Python repositories spanning multiple application domains, including web frameworks, machine learning, generative AI, infrastructure automation, and developer tooling. The selected projects were: *Ansible*, *Anthropic-Skills*, *Browser-Use*, *ComfyUI*, *Django*, *Docling*, *FastAPI*, *Keras*, *Langflow*, *Llama*, *LocalStack*, *MarkItDown*, *PyTorch*, *Stable Diffusion*, *Transformers*, *vLLM*, and *YOLOv*.

After retrieving the repositories, we filtered the code fragments to identify those suitable for inclusion in the study. This filtering step was conducted using a tool developed in this research, named `Scylla`, as described in Section 3.3. At this stage, we exclusively employed a heuristic-based approach using metric-based rules. The purpose of this strategy was to pre-select candidate code fragments with a higher likelihood of containing code smells, thereby improving the balance between positive and negative samples in the dataset. This is particularly important because positive instances of code smells are typically less frequent and more difficult to capture in large-scale repositories.

The six code smells considered in this study are *Large Class* (LC), *Lazy Class* (LzC), *Data Class* (DC), *Long Method* (LoM), *Long Parameter List* (LPL), and *Magic Number* (MN). These smells were selected because they are widely documented in the literature and frequently observed in Python projects [5, 7, 8, 43]. Moreover, to improve the quality and diversity of the dataset, we extracted code fragments at different granularity levels, including classes and methods, ensuring representation across various structural patterns. The resulting candidate fragments were then organized and prepared for the subsequent manual labeling phase.

3.2 Labeling Process

In this phase, we designed a recruitment form to invite Python developers to contribute to the manual labeling of code fragments. Participants were first required to review and accept the ethical consent term (Informed Consent Form – ICF). After providing consent, participants were asked to report their demographic and professional information, including their age, highest level of education, area of expertise, and years of professional experience in software development. These data were used to better understand the profile of the volunteers involved in the labeling process and to support subsequent analyses of participant characteristics.

A total of 37 participants contributed to the manual labeling process. The participant pool exhibited heterogeneous levels of professional experience in software development, ranging from early-career developers (32.1%) to more experienced practitioners (67.9%). Furthermore, the majority of participants reported frequent engagement with programming activities. Specifically, most participants (more than 70%) stated that they program daily and between 3–4 times per week. Overall, these results suggest that most participants maintained continuous interaction with software development tasks, which may contribute to more consistent interpretations of source code fragments and increased reliability throughout the manual labeling process.

Table 1: Manual Labeling Questions

Questions	Description
Which code smell is present in this code fragment?	Select the type of code smell identified in the given code fragment.
Decision Strategy	Describe the reasoning or strategy used to identify the code smell (*e.g.*, structural patterns, complexity indicators, or prior knowledge).
What is the severity level of the identified code smell?	Indicate the severity of the detected code smell (*e.g.*, minor, major, or critical), considering its potential impact on readability, maintainability, and overall code quality.
Severity Justification	Provide a brief explanation supporting the selected severity level, highlighting the factors that influenced the assessment (*e.g.*, size, complexity, or potential maintenance risks).
How difficult was it to classify this code fragment?	Evaluate the perceived difficulty of identifying the code smell (*e.g.*, easy, moderate, hard), reflecting the level of ambiguity, complexity, or clarity of the code fragment.
Difficulty Justification	Provide a concise explanation supporting the difficulty rating, describing the elements that made the detection straightforward or challenging.

After collecting participants' background information, we employed `AnnotAISE`[4], an annotation tool developed by our research group. The selection of this tool was motivated by its ability to randomly distribute code fragments among participants, ensuring unbiased assignment, as well as to automatically manage code fragment allocation. Additionally, it provides built-in mechanisms to assess agreement across different annotations, supporting consistency analysis among annotators.

A questionnaire was created to allow participants to manually label Python code fragments according to the presence or absence of specific code smells. Prior to answering the questionnaire, a video was presented to ensure that all participants had a common understanding of how to use `AnnotAISE`. In addition, a guide was prepared and added to the tool to be available to participants throughout the labeling process, to clarify potential questions and support participants. Table 1 shows the structure of the questionnaire. This step involved a multi-stage labeling process to ensure that the dataset contained representative, high-quality samples that could serve as a reference for training and evaluating automated code smell detection models.

We call our resulting dataset `PyHub-SMELL`.

3.3 `Scylla`

To support evaluation, we employed `Scylla`, a unified tool that enables multi-approach code smell detection. `Scylla` was designed as a modular, event-driven tool that encapsulates the complexity of multi-approach code smell detection into a single cohesive component. Its architecture is grounded in the principles of event-driven microservices, enabling asynchronous communication, fault isolation, and independent evolution of internal components. This architectural choice allows the seamless integration of heterogeneous approaches, including heuristic-based rules, learning models (ML and DL), and language models (SLMs and LLMs), while maintaining a consistent and reproducible execution workflow. `Scylla` internally abstracts a distributed processing pipeline responsible for handling data ingestion, task orchestration, model inference, and result persistence.

In addition to detection, `Scylla` provides built-in support for task management, allowing the monitoring of execution status of each submitted detection, as well as mechanisms for user feedback

[4]https://annotaise.devc.uk/login

on detection outcomes. Moreover, the tool further supports result exploration and reproducibility by enabling the export of detection outputs. Moreover, `Scylla` incorporates interactive data visualization directly into the platform through a dashboard module that facilitates the analysis of detection results, including agreement across approaches at the code fragment level, temporal trends of detected smells, and frequency distributions of smell types. These features collectively enable systematic evaluation, comparison, and interpretation of results across different approaches.

From a technological perspective, `Scylla` leverages widely adopted technologies to ensure reliability and extensibility. The system is implemented in Python, following Domain-Driven Design (DDD) principles [11] to enforce a clear separation of concerns across application, domain, and infrastructure layers. Asynchronous communication is supported through a message broker (*RabbitMQ* [41]), enabling task distribution and decouples request handling from inference execution. Model execution is managed via the *Ollama* framework[5], allowing the orchestration of multiple local open-source models through a unified interface. Additionally, a *PostgreSQL* database [36] is used to persist detection results, ensuring data integrity, traceability, and reproducibility of experiments. By consolidating the outputs of all approaches into a unified storage, the system enables consistent and systematic comparison across heterogeneous approaches. This centralized design allows a fair evaluation under identical conditions, supporting robust cross-approach analysis.

Furthermore, its architecture supports the incremental addition of new approaches and types of code smells, as well as adaptation to other programming languages, with minimal impact on existing components. This flexibility positions `Scylla` not only as a supporting infrastructure for the present study but also as a reusable platform for future research in automated code smell detection. Next, we detail the approaches supported by `Scylla`: *AST-based heuristics*, *learning models* (ML and DL), and *Language Models (LMs)*.

3.3.1 AST-Based Heuristics. In this approach, code fragments are first converted into Abstract Syntax Trees (ASTs), which provide a structured and language-aware representation of the source code. Based on this representation, a set of heuristic rules (see Table 2) is applied to identify the presence of code smells. These rules are defined using software metrics that quantify structural properties at different levels of granularity, including class-level, method-level, and expression-level characteristics. The implementation of these heuristics is grounded in prior studies [5, 7, 48], ensuring alignment with established detection strategies.

Each code smell is associated with specific criteria based on threshold values computed from metrics such as Lines of Code (LOC), Number of Methods (NOM), Number of Attributes (NOA), and cohesion measures (*e.g.*, Lack of Cohesion in Methods—LCOM). For instance, *Large Class (LC)* [5, 7] is identified when the number of lines of code (LOC) exceeds a predefined threshold or when the combination of number of attributes (NOA) and methods (NOM) indicates excessive structural complexity. Similarly, a *Long Method (LoM)* [5, 7] is detected based on the number of lines within a method (MLOC), while *Long Parameter List (LPL)* [5, 7] depends on the number of parameters (PAR) declared in a function or method signature. At a finer level of granularity, expression-level analysis is performed to identify *Magic Numbers (MN)*, defined as numeric literals hard-coded in the source code without explicit semantic meaning, excluding commonly accepted constants (*e.g.*, 0, 1, and -1) [5]. This distinction ensures that the heuristic targets values that may negatively affect readability, maintainability, and long-term code comprehension.

As discussed in Section 2, only a limited number of tools in the literature provide support for classifying Python code smells using AST-based heuristics. Furthermore, these tools are either discontinued, lack active maintenance, or are not open source, which restricts their extensibility and reproducibility in contemporary research and industrial contexts. By incorporating these heuristic models into `Scylla`, we provide an accessible and maintainable implementation of well-established detection strategies. This design enables the use of AST-based heuristics as reliable baseline methods, facilitating systematic comparison with more advanced approaches, such as ML, DL, and LMs.

3.3.2 Learning-Based Approaches. Learning-Based approaches were employed to investigate whether traditional ML and DL approaches could achieve predictive performance comparable to, or surpassing, that of AST-based heuristics and LMs. To this end, the models were trained on our ground-truth dataset (`PyHub-SMELL`) together with software metrics (see Table 3) extracted using `Scylla`. These metrics were designed to capture structural software properties in source code, enabling the transformation of program elements into numerical feature vectors suitable for supervised learning tasks.

To train and evaluate ML models, we employed the PyCaret library[6], a framework designed to streamline the development and experimentation of ML pipelines through automated preprocessing, model selection, and performance evaluation. We evaluated 15 ML algorithms, including: *k-Nearest Neighbor* (kNN) [24], *Logistic Regression* (LR) [24], *Support Vector Machine* (SVM) [24], *Decision Tree* (DT) [21], *Random Forest* (RF) [24], *Extreme Gradient Boosting* (XGB) [24], *Ridge Classifier* (RC) [22], *Light Gradient Boosting* (LGB) [12], *Gradient Boosting* (GB) [17], *AdaBoost* (ADA) [16], *Extra Trees* (ET) [19], *Naive Bayes* (NB) [24], *Linear Discriminant Analysis* (LDA) [4], *Quadratic Discriminant Analysis* (QDA) [39], and *Dummy Classifier* (DuC) [54]. The selection of these algorithms was guided by prior studies reporting commonly adopted ML techniques in the context of defect prediction and code smell detection [44].

For the DL component, we used the *AutoKeras*[7] framework to support automated model search and hyperparameter optimization. We evaluated the *Multi-Layer Perceptron* (MLP) [34] architecture commonly used in software engineering tasks [58]. These architectures were selected due to their ability to capture nonlinear relationships within structured data representations derived from source code metrics.

To ensure reliable performance estimation, the dataset was randomly partitioned per smell into training and test sets using an 80/20 split strategy, with 80% of the data used for model training and 20% reserved for independent evaluation. During training, we used stratified k-fold cross-validation (with $k = 10$) to preserve the distribution of code smell classes across folds and reduce the risk of overfitting. Standard preprocessing steps, including feature

[5]https://ollama.ai/

[6]https://pycaret.readthedocs.io/en/latest

[7]https://autokeras.com/tutorial/multi/

Table 2: AST-Based Heuristics Implemented in Scylla

Code Smell	Level	Criteria	Metric
Large Class (LC) [5, 7]	Class	$(LOC \geq 200) \vee (NOA + NOM) > 40)$	LOC: Lines of Code; NOA: Number of Attributes; NOM: Number of Methods
Lazy Class (LzC) [48]	Class	$((NOM < 5) \wedge (NOA < 5)) \vee (DIT < 2)$	NOM: Number of Methods; NOA: Number of Attributes; DIT: Depth of Inheritance Tree
Data Class (DC) [48]	Class	$(LWMC > 50) \vee (LCOM > 0.8)$	LWMC: Lack of Weighted Method Complexity; LCOM: Lack of Cohesion of Methods
Long Method (LoM) [5, 7]	Function	$MLOC \geq 67$	MLOC: Method Lines of Code
Long Parameter List (LPL) [5, 7]	Function	$PAR > 4$	PAR: Number of Parameters
Magic Number (MN) [5]	Expression	$((val \notin \{0, 1, -1\}) \wedge$ (not ConstantAssignment) $\wedge$ (not ParameterDefault))	NL: Numeric Literal (hard-coded value without semantic definition)

Table 3: Used Software Metrics

Class	Metrics
Raw Metrics	Source lines, logical lines, comment lines, multiline comments, blank lines, and single comment lines
Cyclomatic Complexity	Complexity score, rank, element type, name, line range, class name, and nested method information
Maintainability Index	Maintainability Index value and rank
Halstead Metrics	Halstead operators counts and derived metrics such as vocabulary, length, volume, difficulty, effort, time, and estimated bugs

normalization and handling of missing values, were automatically performed by the PyCaret pipeline. Model performance was evaluated using widely adopted detection metrics, including precision, recall, F1-score, and accuracy (see Section 3.4).

In Section 4.2, we report the top-3 ML models and the top-1 DL model for each code smell type based on their predictive performance. However, all trained models, configuration files, and performance results are available in our supplementary material to support reproducibility and facilitate future comparative studies. Moreover, Scylla enables users to select pre-trained models directly through its interface to new code fragments, compare predictions across different approaches, and evaluate their effectiveness in real-world code review and code comprehension scenarios.

3.3.3 LM-Based Approaches. LM-based approaches were implemented in Scylla to enable the automated detection of code smells using natural language reasoning and contextual understanding of source code. Unlike traditional heuristic or learning-based models that rely primarily on predefined metrics or feature vectors, LMs are capable of analyzing both syntactic and semantic characteristics of code, generating predictions accompanied by natural language explanations. This capability can improve code comprehension and interpretability during code review.

In this study, we evaluated eight different LMs, comprising four SLMs and four LLMs, as summarized in Table 4. Although there is no consensus on the definition of SLMs [18, 27, 29, 52, 53], they are typically defined as models with fewer than 10B parameters. These models were selected to represent different architectural scales and computational requirements, enabling comparison between lightweight and high-capacity models in the context of code smell detection. Although LLMs are capable of handling difficult tasks, their large size and the high cost of running them create major obstacles [42]. These resource requirements often make it difficult to use them in many everyday software tools and real-world systems [23]. In contrast, SLMs have emerged as a viable solution to these challenges, demonstrating that they can match the accuracy of their larger counterparts within specialized areas [47, 52]. With fewer parameters, SLMs significantly reduce hardware and memory

Table 4: Overview of the LMs evaluated in this work

Type	Model	#P	Quant.	Tags	Size	Context	#T
SLM	Qwen2.5-coder	7.62B	Q4_K_M	dae161e27b0e	4.7 GB	32768	4096
	Codegemma	7.62B	Q4_K_M	dae161e27b0e	5.0 GB	6144	2048
	Mistral	7.25B	Q4_0	f974a74358d6	4.1 GB	32768	4096
	Codellama	6.74B	Q4_0	8fdf8f752f6e	3.8 GB	16384	4096
LLM	Codellama	13B	Q4_0	9f438cb9cd58	7.4 GB	16384	4096
	Deepseek-coder-v2	16B	Q4_0	63fb193b3a9b	8.9 GB	32768	4096
	Qwen2.5-coder	14.8B	Q4_K_M	9ec8897f747e	9.0 GB	32768	4096
	Deepseek-R1	14.8B	Q4_0	c333b7232bdb	9.0 GB	32768	4096

requirements for both training and inference, making them an ideal choice for code smell detection.

Prompt engineering mitigates SLMs limitations by guiding model behavior through task-specific instructions [23, 40]. To guide the detection process, prompts were designed using three widely adopted prompting strategies: *zero-shot*, *one-shot*, and *Chain-of-Thought* (CoT) reasoning. For each prompting strategy, we combined source code fragments with contextual information derived from software metrics. We further evaluated two contextual variations: prompts without structural context and prompts enriched with source code metrics extracted from Scylla. In the first variation, the model receives only the textual instruction and the source code snippet, allowing us to assess its ability to identify code smells directly from the code representation. In the second, the prompt additionally includes software metrics, providing explicit structural information that may support detection. This design resulted in a total of six prompt configurations (three prompting strategies $\times$ two contextual variations), which were evaluated across eight models, resulting in a total of 48 variants. Thus, producing a comprehensive set of experimental combinations.

Incorporating metric-based contextual information into prompts enables the enrichment of model inputs with complementary structural features, potentially improving the model's ability to reason about design-level characteristics and detect subtle patterns associated with code smells. This distinction enables us to examine the extent to which syntactic context influences model behavior and predictive performance. The design of these prompts was informed by established literature on software metrics and code quality, including the work of Lanza et al. [26] and Fowler [15].

Although the prompting strategies differ in how they guide model reasoning, they share a structure composed of an instruction, a code fragment, and a standardized JSON output. The main variation lies in whether the prompt includes labeled examples, explicit step-by-step reasoning instructions, and additional structural context derived from AST-based metrics.

The *zero-shot* prompts were designed to elicit direct detections without intermediate reasoning steps. In this setting, the model receives a concise instruction to classify a code snippet based on a target code smell. The *one-shot* prompts follow a similar structure, but additionally include a labeled example to demonstrate the expected reasoning pattern and output format. In turn, the *CoT* prompts introduce an explicit reasoning-oriented formulation, encouraging the model to analyze the code step by step before producing the final detection. All prompts are available in our complementary material.

To ensure fair comparison across models and prompting strategies, all configurations followed a unified prompt structure and task definition, thereby minimizing variability introduced by prompt design. Additionally, deterministic settings (*e.g.*, temperature set to 0.0) were adopted to reduce stochastic variability in model outputs and ensure reproducibility of results. Other parameters, such as context window size, token limits, and standardized output formatting, were carefully configured to maintain consistency across executions. These design choices enabled controlled experimentation, reliable comparison among models, and reproducible evaluation of LM-based approaches.

Within `Scylla`, users can select LM-based detection directly through the system interface. The platform allows users to choose the desired model, prompting strategy, and contextual configuration, enabling flexible experimentation across different settings. Additionally, the system stores both detection results and generated explanations, allowing users to analyze the models' reasoning and compare outcomes across different configurations. This functionality supports exploratory analysis, model comparison, and enhanced code comprehension during practical software maintenance and review activities.

3.4 Data Analysis

This study is guided by two research questions (RQs):

RQ_1 How does LMs performance in Python code smell detection vary across prompt engineering strategies? This question investigates the effectiveness of LMs in detecting Python code smells under different prompt engineering strategies. We evaluated six strategies and employed eight open-source LMs (see Table 4), representing both Small Language Models (SLMs) and Large Language Models (LLMs). The evaluation assesses the models' ability to reliably identify and classify code smell instances without prior fine-tuning. As ground truth, we rely on the manually annotated dataset `PyHub-SMELL` (see Section 3.2), in which code fragments were labeled through a structured human annotation process. This setup enables the measurement of alignment between LM predictions and expert judgments. Overall, this analysis provides a comprehensive understanding of how prompt design choices influence detection performance, robustness, and consistency across models.

RQ_2 How do LMs compare to traditional heuristic, ML and DL approaches in terms of performance for Python code smell detection? This question investigates the extent to which LMs can achieve performance comparable to, or exceeding, that of traditional heuristic, ML, and DL approaches for Python code smell detection. We compared the best-performing LM configurations identified in RQ_1 against both data-driven models (ML/DL) and the AST-based heuristic baseline, which operationalizes the rules described in Table 2. This comparison aims to characterize trade-offs across the predictive performance (accuracy, recall, precision, and F1-score), model size and maintenance effort of different approaches to better understand the practical viability of them for automated code quality assessment and large-scale code analysis.

To answer the research questions, we used `Scylla` to systematically collect and store the detection outputs generated by all supported approaches. Manual labeling data were directly retrieved from the `AnnotAISE` platform, comprising code fragments labeled by human annotators following a structured validation protocol. These datasets were integrated and organized to enable the comparison between automated predictions and human labeling.

A quantitative analysis of model predictions was conducted using widely adopted evaluation metrics, including accuracy, precision, recall, and F1-score. These metrics enabled the assessment of the models' ability to correctly identify code smell instances while balancing false positives and false negatives. All data, scripts, and processing artifacts were curated and structured to ensure transparency and reproducibility of the experimental pipeline.

4 Results

The results are presented and discussed in accordance with the research questions (RQs), as follows.

4.1 Impact of Prompt Engineering Strategies on Language Model Performance (RQ_1)

The results, presented in Table 5, indicate that the effectiveness of prompting strategies depends on both the type of smell and the LM used. The table is organized to facilitate comparison across code smell types, models, and prompting configurations. Each row block corresponds to a specific code smell class (*i.e.*, LoM, LPL, LC, LzC, DC, and MN), while the individual rows within each block represent the evaluated language models. This structure allows the reader to compare how different models perform when classifying the same type of smell.

Across the columns, the table presents the performance obtained under different prompting strategies. Each prompting configuration—namely Zero-shot (Baseline), Zero-shot with context, One-shot, One-shot with context, Chain-of-Thought (CoT), and CoT with context—is represented by a group of three subcolumns reporting Precision, Recall, and F1-score, respectively. Bold values denote the highest F1-score achieved within each smell class, allowing quick identification of the best-performing configuration for that specific detection task.

Overall, the findings show that prompt engineering can significantly influence model performance; however, this impact is neither uniform across models nor consistent across different types of code smells. A closer inspection of Table 5 reveals substantial variability across both smell categories and prompting strategies, reinforcing the importance of selecting configurations tailored to the characteristics of the target smell.

In particular, reasoning-based strategy (CoT) yielded notable improvements for more complex and semantically rich smells, where structural interpretation and contextual reasoning are required. For instance, in the case of *LC*, models such as Deepseek-r1 achieved strong performance, reaching an F1-score of 0.84 under the CoT

Table 5: Performance comparison of LMs using different prompt engineering strategies for code smell detection

Smell	Model	Zero-shot (Baseline)			Zero-shot with context			One-shot			One-shot with context			Chain of Thought (CoT)			CoT with context		
		Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
LoM	Deepseek-r1 14b	0.58	0.61	0.40	0.58	0.62	0.41	0.46	0.49	0.46	0.57	0.59	0.58	0.57	0.62	0.44	0.58	0.64	0.47
	Codegemma 7b	0.55	0.59	**0.48**	0.54	0.57	**0.49**	0.42	0.50	0.45	0.42	0.50	0.45	0.47	0.48	0.47	0.54	0.51	0.50
	Mistral	0.52	0.51	0.23	0.50	0.50	0.19	0.42	0.50	0.45	0.42	0.50	0.45	0.59	0.58	0.31	0.60	0.57	0.29
	Codellama 7b	0.59	0.53	0.20	0.58	0.59	0.35	0.42	0.50	0.45	0.42	0.50	0.45	0.43	0.47	0.45	0.48	0.48	0.48
	Codellama 13b	0.49	0.50	0.48	0.41	0.49	0.45	0.42	0.50	0.45	0.42	0.50	0.45	0.41	0.49	0.45	0.41	0.49	0.45
	Deepseek-coder-v2 16b	0.58	0.63	0.43	0.56	0.58	0.40	**0.61**	0.52	0.51	0.42	0.50	0.45	0.50	0.50	0.49	0.50	0.50	0.50
LPL	Deepseek-r1 14b	0.55	0.55	0.47	0.53	0.53	0.44	0.60	**0.59**	0.48	**0.66**	**0.63**	0.50	0.50	0.50	0.43	0.50	0.50	0.44
	Codegemma 7b	0.46	0.45	0.45	0.42	0.42	0.42	0.34	0.50	0.40	0.34	0.50	0.40	0.34	0.50	0.40	0.50	0.50	0.42
	Mistral	0.62	**0.60**	0.48	0.67	0.54	0.33	**0.74**	0.53	0.46	0.59	0.55	0.55	**0.60**	**0.61**	**0.55**	**0.55**	**0.56**	**0.52**
	Codellama 7b	**0.66**	0.51	0.27	0.67	0.52	0.30	0.34	0.50	0.40	0.34	0.50	0.40	0.34	0.50	0.40	0.34	0.50	0.40
	Codellama 13b	0.41	0.49	0.41	0.33	0.46	0.38	0.34	0.50	0.40	0.34	0.50	0.40	0.34	0.50	0.40	0.34	0.50	0.40
	Deepseek-coder-v2 16b	0.64	0.57	0.40	**0.67**	0.55	0.34	0.33	0.49	0.40	0.46	0.49	0.43	0.44	0.46	0.44	0.47	0.48	0.46
LC	Deepseek-r1 14b	0.72	0.72	0.69	0.76	0.76	0.76	0.39	0.43	0.39	0.62	0.59	0.58	0.76	0.76	0.73	**0.84**	**0.85**	**0.84**
	Codegemma 7b	0.29	0.47	0.36	0.64	0.53	0.46	0.29	0.45	0.35	0.30	0.50	0.37	0.30	0.50	0.37	0.30	0.50	0.37
	Mistral	0.74	0.70	0.65	0.74	0.73	0.69	0.30	**0.50**	0.37	0.30	0.50	0.37	0.73	0.63	0.52	0.73	0.60	0.48
	Codellama 7b	0.68	0.61	0.51	0.61	0.57	0.47	0.30	**0.50**	0.37	0.30	0.50	0.37	0.74	0.73	0.73	0.52	0.52	0.52
	Codellama 13b	0.29	0.48	0.37	0.29	0.45	0.35	0.30	**0.50**	0.37	0.29	0.48	0.37	0.55	0.51	0.41	0.30	0.50	0.37
	Deepseek-coder-v2 16b	0.83	0.75	0.76	0.82	0.83	0.82	0.30	**0.50**	0.37	0.30	0.50	0.37	0.64	0.53	0.46	0.81	0.55	0.47
LzC	Deepseek-r1 14b	0.53	0.55	0.52	0.64	0.74	0.66	0.59	0.69	0.43	0.81	0.81	0.81	0.96	0.67	0.73	**1.00**	**1.00**	**1.00**
	Codegemma 7b	0.61	**0.74**	0.50	0.72	0.79	0.75	0.06	0.50	0.11	0.57	0.52	0.16	0.58	0.62	0.33	0.57	0.55	0.21
	Mistral	0.49	0.48	0.44	0.43	0.48	0.45	0.59	0.67	0.40	0.50	0.50	0.35	**0.98**	0.75	0.82	0.96	0.67	0.73
	Codellama 7b	0.53	0.57	0.44	0.49	0.48	0.31	0.06	0.50	0.11	0.06	0.50	0.11	0.38	0.38	0.17	0.38	0.38	0.17
	Codellama 13b	0.06	0.50	0.11	0.58	0.64	0.37	0.06	0.50	0.11	0.06	0.50	0.11	0.57	0.52	0.16	0.06	0.50	0.11
	Deepseek-coder-v2 16b	0.44	0.50	0.47	**0.96**	0.67	0.73	0.57	0.52	0.16	0.06	0.50	0.11	0.61	0.74	0.50	0.59	0.69	0.43
DC	Deepseek-r1 14b	0.42	0.50	0.46	**0.68**	0.60	**0.62**	0.50	0.51	**0.50**	0.61	**0.65**	**0.62**	0.68	0.60	0.62	**0.70**	**0.70**	**0.70**
	Codegemma 7b	**0.51**	**0.51**	0.45	0.55	**0.62**	0.50	0.04	0.50	0.08	0.58	0.52	0.19	0.66	**0.76**	**0.67**	0.56	0.61	0.48
	Mistral	0.48	0.46	0.39	0.55	0.58	0.53	0.08	0.50	0.14	0.08	0.50	0.14	**0.94**	0.62	0.67	0.42	0.48	0.44
	Codellama 7b	0.35	0.32	0.20	0.35	0.32	0.20	0.08	0.50	0.14	0.08	0.50	0.14	0.08	0.50	0.14	0.08	0.50	0.14
	Codellama 13b	0.08	0.50	0.14	0.08	0.50	0.14	0.08	0.50	0.14	0.08	0.50	0.14	0.08	0.50	0.14	0.08	0.50	0.14
	Deepseek-coder-v2 16b	0.49	0.48	**0.48**	0.42	0.50	0.46	**0.58**	**0.52**	0.19	0.08	0.50	0.14	0.55	0.55	0.55	0.55	0.55	0.55
MN	Deepseek-r1 14b	0.58	**0.60**	**0.55**	0.56	0.56	0.47	0.63	**0.61**	0.49	**0.63**	**0.61**	0.46	**0.63**	**0.63**	**0.52**	0.63	**0.64**	**0.54**
	Codegemma 7b	0.58	0.52	0.49	0.53	0.53	**0.53**	0.35	0.50	0.41	0.35	0.50	0.41	0.35	0.48	0.41	0.57	0.51	0.45
	Mistral	0.64	0.51	0.24	0.64	0.50	0.23	**0.69**	0.52	0.47	0.60	0.55	**0.53**	0.53	0.50	0.25	0.65	0.52	0.27
	Codellama 7b	0.64	0.51	0.24	0.65	0.50	0.23	0.35	0.50	0.41	0.35	0.50	0.41	0.49	0.48	0.46	0.53	0.52	0.35
	Codellama 13b	0.49	0.49	0.49	0.56	**0.57**	0.52	0.35	0.50	0.41	0.35	0.50	0.41	0.52	0.52	0.51	0.44	0.47	0.44
	Deepseek-coder-v2 16b	0.65	0.52	0.26	0.64	0.51	0.23	0.54	0.53	0.42	0.50	0.50	0.36	0.59	0.55	0.39	0.58	0.55	0.40

with context configuration, representing one of the highest results observed across all experiments. Similarly, for *LzC*, the Deepseek-r1 model achieved optimal performance (F1-score of 1.00), indicating perfect detection under reasoning-driven prompting. These results suggest that CoT-based strategies help models decompose structural characteristics into interpretable reasoning steps, improving their ability to detect design-level smells.

Through a qualitative analysis of a representative sample of model-generated outputs, CoT-based approaches also demonstrated clear advantages in terms of interpretability and explanatory depth. In contrast to shorter responses produced by simpler prompting strategies, CoT-based outputs typically presented structured reasoning steps that explicitly described the factors leading to the predicted detection. These explanations often included meaningful insights into potential design violations, such as identifying excessive responsibilities concentrated within a single class or highlighting methods containing an unusually high number of parameters. In several cases, the responses also provided actionable recommendations for refactoring, including suggestions to decompose large classes into smaller, more cohesive units, extract helper methods, or reduce parameter lists through encapsulation techniques. By explicitly referencing structural elements of the code, the generated explanations helped clarify how specific metrics or code characteristics contributed to the detection outcome. This capability is particularly valuable in scenarios where developers need to understand why a given code fragment was flagged as problematic. Such interpretability can reduce uncertainty during code review and facilitate knowledge transfer among team members, especially in large or unfamiliar codebases.

> **Finding 1:** Reasoning-based prompting (CoT) consistently benefits semantically complex smells and generates structured, actionable explanations, offering a significant advantage for real-world adoption.

For code smells such as *LPL* and *MN*, the gains from more sophisticated prompting techniques were less pronounced. In these cases, simpler approaches, such as zero-shot or zero-shot with context, often achieved comparable or even competitive results relative to more complex prompting strategies. For example, the Mistral model obtained competitive performance in LPL detection (F1-score of 0.48 under zero-shot and 0.55 under CoT), indicating only moderate improvement. This behavior suggests that LPL and MN are more closely associated with explicit syntactic patterns, reducing the need for elaborate reasoning. In such scenarios, the overhead introduced by multi-step reasoning prompts may not translate into proportional performance gains.

> **Finding 2:** Simpler smells benefit less from complex reasoning prompts.

Additionally, incorporating software metrics derived from AST as contextual information did not lead to consistent improvements across models. While some scenarios exhibited marginal gains (particularly for smells involving structural complexity), others showed performance degradation. This variability indicates that the inclusion of additional structured data may, in certain cases, introduce noise or increase the cognitive load imposed on the model, rather than providing meaningful guidance. Notably, improvements were

more evident when contextual metrics complemented reasoning prompts, particularly in CoT-based configurations applied to structural smells such as *LC* and *LzC*.

> **Finding 3:** The inclusion of structural metrics as context produces mixed results.

Another important observation is the variability in model sensitivity to prompting strategies. Models such as Deepseek-r1 and Deepseek-coder-v2 demonstrated a higher degree of adaptability to different prompting strategies, resulting in significant performance variations across configurations. For example, Deepseek-coder-v2 achieved strong results across multiple smell types. In contrast, models such as Codegemma and Codellama variants exhibited more stable behavior across configurations, albeit with generally lower performance values. This suggests that the effectiveness of prompt engineering is not solely determined by prompt design but is also strongly influenced by the architectural characteristics and training processes of the models.

> **Finding 4:** Model architecture strongly influences responsiveness to prompting strategies.

A broader analysis of the table also reveals that performance consistency varies substantially across smell categories. Smells such as *LC* and *LzC* generally achieved higher F1-scores across multiple models, indicating that these smells may present clearer structural signatures that are easier for models to identify. In contrast, smells such as *DC* and *MN* exhibited more moderate and variable performance, suggesting greater ambiguity in detection boundaries and potentially higher dependence on contextual interpretation.

From a practical perspective, the results allow the derivation of recommendations of prompting strategies based on smell type:

- *Long Method (LoM)*: CoT prompting is recommended, particularly when combined with contextual information. LoM detection often requires reasoning about method length, internal complexity, and responsibility distribution, which benefits from structured multi-step reasoning.
- *Large Class (LC):* CoT with context is recommended, as reasoning-based prompts significantly improve the detection of structural complexity and excessive class responsibilities.
- *Lazy Class (LzC)*: CoT-based strategies are highly effective, especially when combined with contextual information. These smells benefit from reasoning about class responsibilities and structural relevance.
- *Long Parameter List (LPL)*: Zero-shot or zero-shot with context approaches are generally sufficient, as detection is largely driven by explicit parameter counts and syntactic indicators.
- *Magic Number (MN)*: Lightweight prompting strategies are recommended, since numeric literal detection relies primarily on local syntactic patterns rather than complex semantic interpretation.
- *Data Class (DC)*: Mixed strategies may be beneficial. While CoT improves reasoning in some cases, context-enhanced zero-shot prompts also provide competitive performance.

> **Finding 5:** Smells defined by explicit structural thresholds (*e.g.*, *LPL*, *MN*) tend to benefit from simpler prompts, whereas smells whose classification requires broader reasoning about code structure, organization, and responsibilities (*e.g.*, *LM*, *LC*, *LzC*) benefit more from reasoning-based strategies such as CoT.

Overall, the results demonstrate that prompt engineering plays a fundamental role in the performance of LMs for code smell detection. However, its effectiveness is highly context-dependent, varying according to the complexity of the smell, the nature of the detection task (syntactic versus semantic), and the architectural characteristics of the model.

These findings highlight the importance of carefully selecting and tailoring prompting strategies to maximize the effectiveness of LMs in automated code quality analysis. They suggest that a single prompting strategy is unlikely to be universally optimal, reinforcing the need for adaptive prompt design frameworks capable of dynamically selecting strategies based on the characteristics of the target code smell.

4.2 Comparison Across Approaches (RQ_2)

To answer this research question, we compared the best LM results obtained in Section 4.1 with heuristics, ML, and DL. Table 6 summarises this comparison by reporting, for each code smell class, the best LM configuration, the available heuristic baseline, and the best ML and DL models ranked by F1-score. Overall, the comparison reveals a broadly consistent pattern: ML and DL models outperform the AST-based heuristic baseline across all six smell classes, while LMs outperform the heuristic for most smells, with MN as the main exception. For most smells, traditional ML models trained on tabular code metrics achieved the highest or most stable F1-scores among learned approaches, although important exceptions exist, particularly for LC, LzC, and DC.

> **Finding 6:** Across all six code smell classes, the strongest data-driven baselines outperform the AST-based heuristic baseline, indicating that learned models are better suited than fixed-threshold rules for fragment-level code smell detection.

As shown in Table 6, LMs achieved their strongest results for LC (F1-score up to 0.84), LzC (up to 1.00), and DC (up to 0.70). In these cases, they matched or exceeded the top ML baselines, whose best F1-scores were 0.83 for LC, 0.80 for LzC, and 0.56 for DC. In contrast, ML clearly outperformed LMs for LoM, LPL, and MN, where the best ML F1-scores reached 0.91, 0.80, and 0.82, respectively, while the best LM results were substantially lower, peaking at 0.58, 0.55, and 0.55. These findings suggest that LMs may be particularly competitive for smells that require richer semantic interpretation, whereas metric-based ML models remain stronger for smells that are more directly captured by structural and tabular signals in the present dataset.

> **Finding 7:** LM-based configurations are particularly competitive for LC, LzC, and DC, matching or surpassing the best

Table 6: Performance comparison of the best LM approaches, heuristic, and top-3 ML and DL for code smell detection.

Class	Model	Accuracy	Precision	Recall	F1
Best SLM/LLM					
LoM	Deepseek-r1 14b (1S+Ctx)	0.57	0.57	0.59	0.58
LPL	Mistral (CoT)	0.60	0.60	0.61	0.55
LC	Deepseek-r1 14b (CoT+Ctx)	0.84	0.84	0.85	0.84
LzC	Deepseek-r1 14b (CoT+Ctx)	1.00	1.00	1.00	1.00
DC	Deepseek-r1 14b (CoT+Ctx)	0.84	0.70	0.70	0.70
MN	Deepseek-r1 14b (ZS)	0.56	0.58	0.60	0.55
AST-Based Heuristic					
LoM	Metric-based Rules (Table 3)	0.22	1.00	0.05	0.10
LPL		0.40	0.59	0.37	0.46
LC		0.72	1.00	0.53	0.70
LzC		0.29	0.15	1.00	0.26
DC		0.52	0.10	0.25	0.14
MN		0.61	0.89	0.51	0.65
Top-3 ML Models					
LoM	Random Forest	0.84	0.86	0.96	0.91
	Logistic Regression	0.83	0.84	0.99	0.91
	LightGBM	0.83	0.86	0.96	0.90
LPL	Logistic Regression	0.69	0.68	0.98	0.80
	Ridge Classifier	0.68	0.68	0.98	0.80
	Linear Discriminant Analysis	0.68	0.68	0.97	0.80
LC	Naive Bayes	0.83	0.90	0.79	0.83
	Logistic Regression	0.81	0.86	0.79	0.82
	SVM - Linear Kernel	0.74	0.80	0.89	0.80
LzC	Extra Trees	0.95	0.77	0.87	0.80
	SVM - Linear Kernel	0.95	0.67	0.73	0.69
	Logistic Regression	0.95	0.67	0.67	0.67
DC	Logistic Regression	0.91	0.60	0.53	0.56
	Extra Trees	0.90	0.60	0.57	0.56
	Random Forest	0.90	0.57	0.53	0.53
MN	Logistic Regression	0.71	0.72	0.95	0.82
	Ridge Classifier	0.70	0.72	0.95	0.81
	Linear Discriminant Analysis	0.70	0.72	0.95	0.81
Best DL (Balanced Accuracy)					
LoM	AutoKeras	0.60	0.86	0.83	0.84
LPL	MLP	0.52	0.69	0.67	0.64
LC	MLP	0.85	0.95	0.74	0.83
LzC	MLP	0.80	0.33	1.00	0.47
DC	MLP	0.67	0.38	0.60	0.42
MN	MLP	0.63	0.79	0.67	0.71

ML baselines for these smells, which suggests that LMs are especially effective when richer semantic interpretation is required.

The heuristic baseline offers a complementary point of comparison across all six smell classes. The heuristics reported here are implemented as the AST-based detection component of our framework, applying the fixed structural thresholds defined in Table 2 directly to code fragments, without any learning step.

For LoM, the heuristic achieved F1 = 0.10, driven by near-zero recall (0.05) alongside perfect precision (1.00): the MLOC ≥ 67 threshold is almost never met by the method-level fragments in this dataset, despite many being labeled as LoM by expert annotators. This indicates that the fixed MLOC threshold is too conservative for the fragment-level dataset: many snippets labeled as LoM by expert annotators do not satisfy the hardcoded length criterion, causing the heuristic to severely under-detect this smell. For LPL, the heuristic achieves only moderate performance (F1 = 0.46) and remains well below both ML and DL baselines. For LC, the heuristic achieved F1 = 0.70, with perfect precision (1.00), but moderate recall (0.53). The high precision indicates that instances flagged by the heuristic are almost invariably true positives; however, it misses roughly half of the expert-labeled cases, placing this result below both the best LM baseline (F1 = 0.84) and the best ML baseline (F1 = 0.83). For LzC, the heuristic exhibited the opposite pattern: recall of 1.00 combined with very low precision (0.15), yielding F1 = 0.26. This behavior reflects a structural limitation of threshold-based detection on isolated fragments: the DIT < 2 component of the LzC criterion fires almost unconditionally, because the inheritance depth of a class cannot be determined without access to the full project hierarchy, which is unavailable in snippet-level analysis. As a result, the heuristic substantially underperforms both the best LM (F1 = 1.00) and the best ML baseline (F1 = 0.80). For DC, the heuristic achieved F1 = 0.14, reflecting the limited structural signal available from isolated class fragments under the LWMC and LCOM criteria. For MN, the heuristic achieved F1 = 0.65, surpassing the best standalone LM result (F1 = 0.55), but still remaining below the best ML (F1 = 0.82) and DL (F1 = 0.71) baselines. Overall, these results confirm that fixed structural rules yield inferior and less adaptable predictive performance than learned approaches under the present fragment-level setting, while also showing that self-contained syntactic smells, such as MN, can still be partially captured by hardcoded rules.

Finding 8: The heuristic baseline is influenced by fragment-level analysis, as some rules rely on structural context not fully preserved in isolated snippets, which may affect precision and recall across smell classes.

A similar pattern is observed when LMs are compared against DL. According to Table 6, the best DL results in the current study were obtained by MLP and AutoKeras, with F1-scores of 0.84 for LoM, 0.64 for LPL, 0.83 for LC, 0.47 for LzC, 0.42 for DC, and 0.71 for MN. Relative to these baselines, LMs were stronger for DC and LzC, and marginally superior for LC (LM F1 = 0.84 vs. DL F1 = 0.83). Conversely, DL outperformed LMs for LoM, LPL, and MN. Even so, DL did not surpass the best ML baselines in the present setup for any smell class. This indicates that, although DL can be competitive, especially for LC and LoM, it does not constitute the strongest non-LM family overall in our results. Moreover, the DL comparison is methodologically narrower, since only MLP and AutoKeras produced executable results under the adopted protocol, while other architectures were either not run for small-sample smells or excluded because the current inputs are tabular numeric metrics rather than naturally sequential representations.

From a practical perspective, these findings point to complementary trade-offs among the approaches. The heuristic approach, despite implementing well-established detection rules, consistently underperforms both ML and DL baselines across all smell classes, and also underperforms LMs for most smells, with MN as the exception. ML models emerge as the most reliable learned family, consistently achieving the highest F1-scores on the tabular metric-based features of the present dataset. LMs, in turn, can be applied without task-specific retraining and, as discussed in Section 4.1, may also provide natural-language justifications that are potentially useful to developers. However, their predictive performance

is more sensitive to prompt design and less consistently strong across smells. DL can be competitive, but its gains over ML are not evident under the current tabular formulation.

> **Finding 9:** ML shows the most reliable predictive performance on tabular features, DL remains competitive in selected cases within the current setup, and LMs provide competitive detection with potential natural-language justifications.

Overall, the strongest data-driven approaches consistently outperform the AST-based heuristic baseline across all six smell classes. Among learned approaches, LM-based configurations were particularly strong for LC, LzC, and DC, matching or exceeding the best ML baselines for these smells, while ML remained the strongest family for LoM, LPL, and MN, reflecting the effectiveness of tabular metric-based features for structurally well-defined smells. LMs are a competitive and, in some smell classes, highly effective alternative, particularly for semantically richer smells such as LzC and LC, but they do not consistently outperform ML approaches across all smell types. DL remains competitive in specific cases but is methodologically narrower than ML in the current setup.

4.3 Threats to Validity

The present analysis was conducted within the context of the Python programming language, which may influence the extent to which the findings generalize to other settings. Given Python's widespread adoption, the results are expected to be representative; however, further investigation is warranted to examine whether the observed patterns remain consistent across different programming languages. Moreover, the study considered a specific set of code smells—LoM, LPL, LC, DC, LzC, and MN—which, while comprehensive, defines a particular analytical scope. As a direction for future work, expanding this set to include additional code smells and varying levels of abstraction may provide a more comprehensive assessment of model performance and further enhance the generalizability of the findings.

This study considered six prompting strategies—one-shot, zero-shot, and chain-of-thought (combined with and without context)—to guide the models in classifying the analyzed code snippets. Although these approaches are well-established in the literature, this set of strategies may not fully encompass the spectrum of LMs inferential capabilities. In this sense, future investigations may explore a broader diversity of prompting strategies to provide a more comprehensive characterization of model behavior in the task of code smell detection. Additionally, the models evaluated in this study—ranging from 6.14B to 14.8B parameters may limit the ability to fully capture long contexts and complex program structures, particularly when compared to larger-scale models. However, they present a balance between representational capacity and computational efficiency, supporting their feasibility for practical deployment scenarios.

5 Conclusion and Future Work

This paper presented a protocol for evaluating automated approaches to code smell detection, encompassing AST-based heuristics, Machine Learning (ML), Deep Learning (DL), and Language Models (LMs) under a variety of prompt engineering strategies. Unlike prior studies, which typically focus on a limited set of smells or a single methodological approach, our work investigated several approaches and six widely recognized code smell classes: Long Method (LoM), Long Parameter List (LPL), Large Class (LC), Lazy Class (LzC), Data Class (DC), and Magic Number (MN). In addition, we proposed `PyHub-SMELL`, a manually annotated dataset constructed through a rigorous human-driven validation process, designed to provide a reliable ground truth for training and evaluating detection models. We also introduced `Scylla`, a unified and extensible tool that integrates multiple detection approaches within a single environment. The tool supports systematic experimentation, facilitates reproducible analysis, and enables practitioners and researchers to compare different techniques under consistent conditions.

Our results show that prompt engineering has a substantial impact on LM performance, particularly for semantically richer smells. Reasoning-based strategies, such as Chain-of-Thought, were especially effective for smells like LC and LzC, whereas simpler prompts remained competitive for more syntactically explicit smells, such as LPL and MN. In the cross-approach comparison, learned approaches consistently outperformed the AST-based heuristic baseline, indicating the limitations of fixed-threshold rules when applied to isolated code fragments. Among the learned models, ML achieved the most stable overall performance, especially for structurally well-defined smells, while LMs proved highly competitive for smells requiring richer semantic interpretation, in some cases matching or surpassing ML baselines. DL remained competitive in selected cases, although it did not consistently exceed ML under the current tabular setting.

Overall, our findings show that no single approach is universally optimal across all smell types. Instead, the effectiveness of automated code smell detection depends on the interaction between smell complexity, model approach, and prompt design. These results reinforce the importance of adaptive and context-aware strategies for code quality assessment and highlight the potential of combining predictive effectiveness with explainability in software engineering tools. Together, the proposed protocol, dataset, and tool provide a robust foundation for advancing research in automated code smell detection and for bridging the gap between experimental evaluation and practical adoption.

As future work, we plan to expand the evaluation to additional code smell categories, investigate hybrid strategies that combine the strengths of heuristics, ML, and LMs, and further investigate the grounding and actionability of model-generated explanations. We also intend to explore broader datasets and project-level contexts to better understand how these approaches behave in more realistic software maintenance settings.

Artifact Availability

The artifacts are available at https://zenodo.org/records/20054026.

Acknowledgments

This research was partially funded by the Brazilian funding agencies CAPES/COFECUB (Grant 88881.879016/2023-01 and Ma1036/24), CNPq (Grant 406089/2025-6), FAPESP (Grant 2023/00811-0), and FAPEMIG (Grant APQ-01488-24). We also acknowledge the Brazilian company Stone[8] for their financial support.

[8] https://www.stone.com.br/

References

[1] 2025. Anonymous.

[2] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering (EMSE)* 21 (2016), 1143–1191.

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

[4] S. Balakrishnama and A. Ganapathiraju. 1998. Linear discriminant analysis-a brief tutorial. *Institute for Signal and information Processing* 18, 1998 (1998), 1–8.

[5] Aryan Boloori and Tushar Sharma. 2025. DPy: Code Smells Detection Tool for Python. In *Proceedings of the 22nd International Conference on Mining Software Repositories (MSR 2025) – Data and Tool Showcase Track*. Ottawa, Canada. https://2025.msrconf.org/details/msr-2025-data-and-tool-showcase-track/2/DPy-Code-Smells-Detection-Tool-for-Python

[6] Joanne Carneiro, Jessica Ribas, Amanda Santana, Eduardo Figueiredo, and Juliana Alves Pereira. 2025. Improvmlcq: a feature-enriched dataset for advancing code smell detection. In *Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*. SBC, 13–23.

[7] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting code smells in Python programs. In *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 18–23.

[8] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 18–23. doi:10.1109/SATE.2016.10

[9] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 560–564. doi:10.1109/MSR52588.2021.00074

[10] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621.

[11] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, Boston, MA, USA. https://www.domainlanguage.com/ddd/ First Edition.

[12] Junliang Fan, Xin Ma, Lifeng Wu, Fucang Zhang, Xiang Yu, and Wenzhi Zeng. 2019. Light Gradient Boosting Machine: An efficient soft computing model for estimating daily reference evapotranspiration with local and external meteorological data. *Agricultural water management* 225 (2019), 105758.

[13] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 1–12.

[14] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 1037–1039.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA. https://martinfowler.com/books/refactoring.html

[16] Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

[17] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[18] Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. 2023. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning*. PMLR, 10421–10430.

[19] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine learning* 63 (2006), 3–42.

[20] Ruchin Gupta, Narendra Kumar, Sunil Kumar, and Jitendra Kumar Seth. 2024. Unsupervised Machine Learning for Effective Code Smell Detection: A Novel Method. *Journal of Communications Software and Systems (JSS)* 20, 4 (2024), 307–316.

[21] Aurélien Géron. 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. (2019).

[22] Arthur E Hoerl and Robert W Kennard. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics* 12, 1 (1970), 55–67.

[23] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 8 (2024), 1–79.

[24] Wenhua Hu, Lei Liu, Peixin Yang, Kuan Zou, Jiajun Li, Guancheng Lin, and Jianwen Xiang. 2023. Revisiting" code smell severity classification using machine learning techniques". In *47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 840–849.

[25] Nasraldeen Alnor Adam Khleel and Károly Nehéz. 2024. Improving accuracy of code smells detection using machine learning with data balancing techniques. *The Journal of Supercomputing* 80, 14 (2024), 21048–21093.

[26] Michele Lanza and Radu Marinescu. 2006. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer.

[27] Jooyoung Lee, Fan Yang, Thanh Tran, Qian Hu, Emre Barut, and Kai-Wei Chang. 2024. Can small language models help large language models reason better?: LM-guided chain-of-thought. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. 2835–2843.

[28] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* 47, 9 (2019), 1811–1837.

[29] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. 2024. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. In *Forty-first International Conference on Machine Learning*.

[30] Lech Madeyski and Tomasz Lewowski. 2020. Mlcq: Industry-relevant code smell data set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 342–347.

[31] Lech Madeyski and Tomasz Lewowski. 2023. Detecting code smells using industry-relevant data. *Information and Software Technology (IST)* 155 (2023), 107112.

[32] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, USA.

[33] Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, USA.

[34] Fionn Murtagh. 1991. Multilayer perceptrons for classification and regression. *Neurocomputing* 2, 5 (1991), 183–197. doi:10.1016/0925-2312(91)90023-5

[35] Henrique Gomes Nunes, Amanda Santana, Eduardo Figueiredo, and Heitor Costa. 2024. Tuning Code Smell Prediction Models: A Replication Study. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC)*. 316–327.

[36] Regina O. Obe and Leo S. Hsu. 2017. *PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database* (3 ed.). O'Reilly Media.

[37] Ollama. 2026. Ollama. https://ollama.com/ Accessed on: May 2, 2026.

[38] Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. 2017. Investigating code smell co-occurrences using association rule learning: A replicated study. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. 8–13. doi:10.1109/MALTESQUE.2017.7882010

[39] Yingli Qin. 2018. A review of quadratic discriminant analysis for high-dimensional data. *Wiley Interdisciplinary Reviews: Computational Statistics* 10, 4 (2018), e1434.

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[41] Gavin M. Roy. 2017. *RabbitMQ in Depth*. Manning Publications, Shelter Island, NY, USA.

[42] Ahmed R Sadik and Siddhata Govind. 2025. Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 969–975.

[43] Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *PeerJ Computer Science* 9 (2023), e1370. doi:10.7717/peerj-cs.1370 https://doi.org/10.7717/peerj-cs.1370.

[44] Geanderson Santos, Amanda Santana, Gustavo Vale, and Eduardo Figueiredo. 2023. Yet Another Model! A Study on Model's Similarities for Defect and Code Smells. In *Fundamental Approaches to Software Engineering*. 282–305.

[45] Karthik Shivashankar and Antonio Martini. 2025. PyExamine: A Comprehensive, Un-Opinionated Smell Detection Tool for Python. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 763–774. doi:10.1109/MSR66628.2025.00114

[46] Luciana Lourdes Silva, Janio Rosa da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) *(ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 400–406. doi:10.1145/3674805.3690742

[47] Shreyas Subramanian, Vikram Elango, and Mecit Gungor. 2025. Small Language Models (SLMs) Can Still Pack a Punch: A survey. arXiv:2501.05465 [cs.CL] https://arxiv.org/abs/2501.05465

[48] Fatma Neda Topuz and David A. Umphress. 2023. Do Bad Smells Lead to Defects?. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*. 75–81. doi:10.1109/ICSTE61649.2023.00020

[49] Adam Tornhill and Markus Borg. 2022. Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*. 11–20.

[50] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R Basili. 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM Sigplan Notices* 34, 10 (1999), 47–56.

[51] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*. 329–331.

[52] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, et al. 2025. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. 87 pages.

[53] Yuling Wang, Changxin Tian, Binbin Hu, Yanhua Yu, Ziqi Liu, Zhiqiang Zhang, Jun Zhou, Liang Pang, and Xiao Wang. 2024. Can small language models be good reasoners for sequential recommendation?. In *Proceedings of the ACM Web Conference 2024*. 3876–3887.

[54] Stewart W Wilson. 2002. Classifiers that approximate functions. *Natural Computing* 1, 2 (2002), 211–234.

[55] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1345–1357.

[56] Pravin Singh Yadav, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2024. Machine learning-based methods for code smell detection: a survey. *Applied Sciences* 14, 14 (2024), 6149.

[57] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. 242–251.

[58] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* 54, 10s, Article 206 (Sept. 2022), 73 pages. doi:10.1145/3505243

[59] Beiqi Zhang, Peng Liang, Xin Zhou, Xiyu Zhou, David Lo, Qiong Feng, Zengyang Li, and Lin Li. 2024. A comprehensive evaluation of parameter-efficient fine-tuning on code smell detection. *ACM Transactions on Software Engineering and Methodology* (2024).