Code Smell Classification in Python: Are Small Language Models Up to the Task?

Igor Soares de Oliveira
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
isoliveira@inf.puc-rio.br

Joanne Carneiro
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
jribeiro@inf.puc-rio.br

Jessica Ribas
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
jribas@inf.puc-rio.br

Juliana Alves Pereira
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
juliana@inf.puc-rio.br

ABSTRACT

Code quality is essential for maintainable and evolvable software systems. Traditional code smell detection tools rely on AST-based techniques and metric-driven heuristics, which, while effective, often lack interpretability and require specialized knowledge. This paper investigates the use of Small Language Models (SLMs) for classifying two widely studied code smells—Long Method and Long Parameter List—in Python codebases. Unlike Large Language Models (LLMs), SLMs offer lower latency and reduced computational cost, making them suitable for deployment in resource-constrained environments. We systematically compare the performance of SLMs with traditional Machine Learning (ML) and Deep Learning (DL) models, and the AST-based DPy tool. We also analyze the role of prompt engineering techniques—zero-shot and chain-of-thought—in enhancing SLM performance. Our evaluation considers precision, recall, F1-score, and processing time, using a custom event-driven dataset designed for code classification. Results show that SLMs achieve competitive accuracy with improved interpretability. Additionally, we release an annotated dataset comprising classifications from all approaches. This work provides new insights into lightweight, explainable, and practical methods for automated code quality assessment, and supports the broader adoption of SLMs in software engineering.

KEYWORDS

Software Quality, Code Smells, Python, Small Language Models

1 Introduction

Ensuring high code quality is essential for the long-term success of any software project. Fowler et al. [8] describe the term code smell as symptoms in source code that, while not strictly erroneous, often reflect deeper design flaws or structural issues. Code smells accumulate technical debt, increase the risk of defects, and slow down development teams [21]. Refactoring—the process of restructuring existing code without changing its external behavior—is the primary strategy for addressing code smells. However, its effectiveness depends heavily on the accurate and timely identification of these smells.

Two pervasive and impactful code smells are Long Method (LM) and Long Parameter List (LPL). An LM bundles multiple responsibilities into a single routine, making it difficult to comprehend, test, and reuse; refactoring LM by the *Extract Function* or moving logic into well-named helper methods typically restores clarity and modularity [8]. An LPL burdens callers and maintainers with a proliferation of arguments—often interdependent or referring to the same conceptual entity—thereby increasing cognitive load; techniques like *Introduce Parameter Object* or *Preserve Whole Object* can collapse related parameters into coherent abstractions [8, 22]. Detecting these smells early is critical: manual inspection does not scale, and traditional metric-based tools often lack context-sensitive explanations and struggle to adapt to evolving codebases.

Recent advances in language modeling offer a promising alternative. Small Language Models (SLMs)—defined here as pre-trained neural models under 8 billion parameters—bridge the gap between heavyweight Large Language Models (LLMs) and classic static analysis tools. With optimized architectures and curated data, SLMs can deliver performance comparable to LLMs while operating with lower inference latency, reduced hardware requirements, and improved on-device privacy guarantees [35, 38]. Unlike Abstract Syntax Tree (AST) based detectors, SLMs can generate natural-language explanations alongside binary smell classifications, augmenting interpretability and aiding developers in understanding the root causes of detected smells.

In this work, we investigate whether SLMs are *"up to the task"* of identifying LM and LPL smells in Python code. Our contributions are fourfold: (1) We design and implement an event-driven API that integrates multiple SLMs for on-demand code smell classification, embedding prompt-engineering techniques—zero-shot and Chain-of-Thought—to elicit precise and reasoned model outputs. (2) We benchmark SLM performance against both AST-metric tools (e.g., the DPy analyzer [2]) and traditional Machine Learning (ML) models (*e.g.*, Decision Trees, Random Forests, Gradient Boosting) and Deep Learning (DL) models (*e.g.*, Multi-Layer Perceptron, Long Short-Term Memory, AutoKeras). (3) We build a large, labeled dataset of Python functions annotated for LM and LPL smells containing the classifications produced by each approach (SLMs, ML, DL, DPy and PySmell), with 230,829 records, enabling experiment replication and supporting future comparative research in the field. We also make available multiclass ML and DL models for code smell classification. (4) Implementation of a web-based frontend application aimed at providing users with an intuitive interface for submitting and classifying Python code snippets.

We demonstrate that SLMs achieve competitive classification metrics, with F1-scores up to 0.87 for LM and 0.59 for LPL using the Mistral model with Chain-of-Thought prompts. Moreover, it also provides actionable explanations that lower the barrier to refactoring for developers relatively new to code-quality concepts. Moreover, our results show that while classical AST-based approaches still hold an edge in raw accuracy (*e.g.*, DPY's F1-score of 0.97 for LM), SLMs offer a compelling balance between detection performance, interpretability, and operational efficiency. Moreover, prompt engineering emerges as a key factor: Chain-of-Thought

strategies consistently outperform zero-shot baselines by guiding models through multi-step reasoning about code structure. Finally, we release our dataset and evaluation framework to support reproducibility and stimulate further research into hybrid static–dynamic analysis techniques for sustainable software development.

2 Related Work

In the literature, several studies address the topic of code quality, with a predominant focus on languages such as Java, C, C++, and C# (*e.g.* [23]). Some of these studies employ tools like inFusion [6, 28], JDeodorant [6, 28, 36], PMD [6, 28], and JSpIRIT [6, 28, 37], which can be integrated into IDEs to support code analysis. Additionally, AST-based techniques for code smell detection in Python have been explored, such as the PySmell [3] and DPY [2] tools. Furthermore, several studies [13, 30, 45] use ML and DL techniques. However, despite their effectiveness, these approaches generally require a significant level of technical expertise to be successfully applied. Tools that integrate with development environments (IDEs) often demand complex configuration procedures and in-depth familiarity with the specific environment. Similarly, ML and DL approaches typically require access to labeled datasets, model training, hyperparameter optimization, and rigorous validation processes to ensure acceptable performance. These requirements can pose substantial barriers to broader adoption, particularly among professionals without specialized knowledge in these approaches or those who are at the early stages of their careers in software development.

LLMs have recently gained attention as a promising tool for detecting code smells. A notable contribution in this area is the empirical study by Silva et al. [33]. This study investigates the use of ChatGPT to classify four common types of code smells (Blob, Data Class, Feature Envy, and Long Method) in Java programs. The authors evaluated the model's performance using both generic and specifically crafted prompts, finding that detailed prompts significantly improved detection accuracy. The study also revealed that ChatGPT is particularly effective in identifying smells with higher severity levels, achieving an F1-score of up to 0.52. These findings not only reinforce the potential of LLMs in software quality analysis but also provide us with valuable methodological insights into prompt design for code analysis tasks.

Wu et al. [41], proposed iSMELL, an approach that combines LLMs with specialized tools for code smell detection and refactoring in Java. iSMELL employs a Mixture of Experts (MoE) architecture to dynamically select the most appropriate set of tools for identifying different types of code smells, such as Refused Bequest, God Class, and Feature Envy. Empirical evaluation demonstrated that iSMELL achieved an average increase of 35.05% in F1-score for smell detection tasks in comparison to standalone expert tools. The study highlights that integrating specialized tools with LLMs offers a scalable and effective approach to maintaining code quality.

In contrast to previous studies, this work proposes a distinct approach to the task of code smell classification by employing two prompting strategies, zero-shot and Chain-of-Thought, with and without additional AST-based context. This methodological choice enables a more controlled and interpretable analysis of model behavior. Furthermore, while most existing studies focus on languages such as Java, C, C++, and C#, as previously noted, this study targets the Python language, expanding the scope of applicability and contributing to a more diverse set of contexts explored in the literature. In addition, this work adopts the use of SLMs, which are more accessible and cost-effective than LLMs, thereby facilitating their adoption in academic settings, industrial environments, and by users with limited computational resources. The proposed architecture, structured around modular microservices, was designed to support both the replication of experiments and the extension of future research. Finally, we plan to explore aspects of model explainability, including automated refactoring suggestions for the classified code samples, thus contributing to improvements in software quality and in the development of practitioners' competencies related to code quality, as evidenced by the provided artifacts.

3 Study Design

This section presents our study design, structured as presented in Figure 1 and detailed as follows.

3.1 Code Extraction and Data Processing

In this phase, we used the PySmell dataset[1]. It provides key properties that enable the extraction of source code from eight projects, such as NLTK, Django, Boto, Tornado, Ansible, Matplotlib, NumPy, and SciPy. We used the fields `subject`, `tag`, and `file` to identify the corresponding repositories, determine the specific version of each project, and locate the files that were manually annotated by experts. After identifying the official GitHub repositories, an extraction algorithm was developed to retrieve the methods and classes labeled in the PySmell dataset. This algorithm relies on attributes such as `line` and `file` to ensure that the extracted code precisely matches the method or class starting at the specified line. Once the correct file and line number are located, the code segment is extracted into a dedicated folder, and the resulting file is named following the pattern `line_filename_method/class_name`. Additionally, a new CSV file is generated containing only the code elements that were successfully retrieved from the original repositories (Figure 1 – Phase 1). This ensures that only code samples manually labeled in PySmell and reliably matched in the source projects were considered in this study, while unmatched entries were excluded (see [34] for more information). It is important to note that this study focused exclusively on the LM and LPL code smells due to compatibility constraints between the PySmell dataset and the DPy tool. These two smell types were the only ones consistently supported and labeled across both resources, thereby enabling consistent and direct comparisons.

3.2 Use of DPY

DPy [3] is a code smell detection tool specifically designed for Python projects and design-level code smells using heuristics based on metrics extracted from the Abstract Syntax Tree (AST). Although not open source, it is freely usable and allows results to be exported in CSV or JSON formats (Figure 1 – Phase 2).

The tool computes object-oriented metrics—such as LOC, NOM, NOPM, and CC—that assess structural aspects like size, complexity, and cohesion. In this study, we use these metrics both to support ML/DL classification and to inform prompt engineering for SLMs.

[1] https://github.com/chenzhifei731/Pysmell

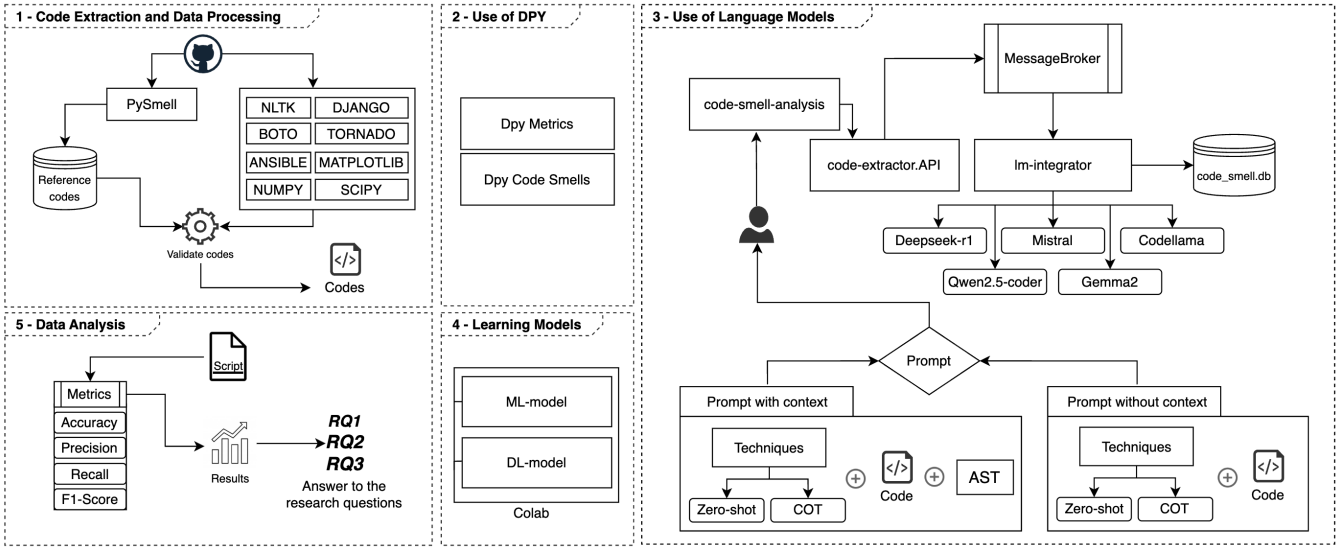

Figure 1: Code Smell Classification System Based on Language Models

DPy detects 19 code smells, including the LM and LPL smells analyzed in this work. These are identified through threshold-based rules applied to code metrics. We adopt DPy as a baseline to compare its performance with ML, DL, and SLM-based methods (Section 4).

3.3 Use of Language Models

The architecture was developed following microservice and event-driven standards, as illustrated in Figure 1 (Phase 3). This architectural style leverages events to trigger actions and enable communication between decoupled services [4], promoting modularity and scalability. It involves three core components [16]: producers, routers, and consumers—where events are emitted, routed, and processed. This design allows independent deployment and evolution of services, supporting high availability and real-time processing, while facilitating maintainability and system extensibility over time.

Furthermore, the adoption of a microservice-based architecture was motivated by the need for flexibility and scalability. This model allows new types of code smells to be added incrementally, with minimal impact on existing services. As a result, the system can be progressively evolved to support different categories of smells while preserving modularity and facilitating maintenance and extensibility of the solution.

The *code-smell-analysis* application serves as a front-end interface for code submission, prompt configuration, SLM selection, and extraction type definition. It guides the user throughout the process, enhancing transparency and usability.

Operating within an anti-corruption layer (ACL) [32], the *code-extractor.api* acts as an event producer, receiving the contract from *code-smell-analysis*, extracting structural elements (classes/methods) via AST [7], and dispatching events to the router in accordance with the output contract. Further implementation details, including message formats, component interactions, and configuration examples, are available in our supplementary material [34].

The *lm-integrator* application operates as an event consumer, processing messages using SLMs integrated via Ollama [27] or RESTful APIs [12, 24]. For this study, five SLMs were evaluated: Deepseek-r1, Qwen2.5-coder, Mistral, Gemma2, and Codellama (see Table 1). Results are persisted in a relational database to support further analysis and reproducibility.

Table 1: Overview of the SLMs evaluated in this work

Model Name	Parameters	Provider	Context	Quantization	Tags	Model Size
Deepseek-r1	1.5 billion	Ollama	128K	Q4_K_M	a42b25d8c10a	1.1 GB
Qwen2.5-coder	1.5 billion	Ollama	32K	Q4_K_M	6d3abb8d2d53	986 MB
Mistral	7.25 billion	Ollama	32K	Q4_0	f974a74358d6	4.1 GB
Gemma2	2.61 billion	Ollama	8K	Q4_0	8ccf136fdd52	1.6 GB
Codellama	6.74 billion	Ollama	16K	Q4_0	8fdf8f752f6e	3.8 GB

3.3.1 Prompt Design. Aiming to analyze the impact of incorporating syntactic information on the performance of language models, this study proposes two variations for each prompting strategy (zero-shot and CoT) [19, 39]. The selection of these approaches is based on the assumption that zero-shot enables the evaluation of the model's ability to respond without prior conditioning, whereas CoT allows for more descriptive and reasoning-driven responses. For each strategy, two variations are proposed: with and without the use of software metrics extracted from the AST. This methodological distinction enables the evaluation of how the inclusion of explicit structural data influences the accuracy and robustness of code smell identification.

In the prompt *without context* configuration, prompts were constructed without the inclusion of structural information derived from the AST. The model receives only a concise instruction requesting the classification of a code snippet based on the presence of a specific code smell. This approach aims to evaluate the intrinsic ability of the language model to interpret and identify undesirable patterns solely from the textual representation of the source code. The prompt structure in this variation follows the format: textual

instruction + code snippet + output formatted in JSON with the corresponding code smell.

Conversely, in the prompt *with context* variation, prompts were enriched with structural metrics extracted from the AST, providing the model with additional syntactic information that may support the inferential process. The inclusion of this structural context aims to examine the extent to which the explicit availability of code characteristics influences the accuracy and effectiveness of automated code smell classification. The prompt structure in this configuration follows the format: textual instruction + code snippet + AST metrics + output formatted in JSON with the corresponding code smell.

Zero-shot

Prompt:
Classify the code as smelly or non-smelly
Code: <add code>
[Metric: <add AST metric>] (only used for Prompt with context)
return the result as one of the following JSONs: {"smell_type":"smelly"} OR {"smell_type":"non-smelly"} AND {"explanation":"your explication about this code"}

Chain of Thought - LM Example

Prompt:
LMs make code difficult to understand, test, and maintain. When a method grows too large, it can exhibit high complexity, excessive conditional statements, nested loops, and multiple responsibilities, violating the Single Responsibility Principle (SRP). Additionally, LMs often mix different levels of abstraction, making the code harder to read and reuse. Changes in this type of code are more prone to errors, affecting multiple parts of the system. Based on these characteristics, the provided code may show signs of this code smell if it has too many lines, a high level of coupling, excessive parameters or local variables, and performs multiple operations that could be extracted into smaller, more cohesive functions. Does the provided code exhibit signs of a LM?
Code: <add code>
[Metric: <add AST metric>] (only used for Prompt with context)
Let's think step by step.
return the result as one of the following JSONs: {"smell_type":"smelly"} OR {"smell_type":"non-smelly"} AND {"explanation":"your explication about this code"}

3.4 Learning Models

Learning models were employed to assess whether their performance could achieve results equal to or superior to those obtained by SLMs (Figure 1 – Phase 4). Accordingly, the models were trained using Dpy metrics to evaluate the predictive performance of different ML and DL algorithms in detecting code smells. To train the models, a multilabel classification approach was adopted as the target strategy.

3.4.1 Code Smell Classification with ML Models. ML models enhance the effectiveness of code smell classification by leveraging project-related metrics [18, 25, 43]. They are capable of simultaneously analyzing multiple metrics and detecting correlations among them, allowing for the identification of real patterns associated with code smells. To train and execute the ML models, we employed the PyCaret library[2]. For this study, we selected 15 ML algorithms, namely: *k-Nearest Neighbor* (kNN) [17], *Linear Regression* (LR) [17], *Support Vector Machine* (SVM) [17], *Decision Tree* (DT) [14], *Random Forest* (RF) [17], *Extreme Gradient Boosting package* (XGB) [17], *Ridge Classifier* (RC) [15], *Light Gradient Boosting* (LGB) [5], *Gradient Boosting* (GB) [10], *Ada Boost* (ADA) [9], *Extra Trees* (ET) [11], *Naive Bayes* (NB) [17], *Linear Discriminant Analysis* (LDA) [1], *Quadratic Discriminant Analysis* (QDA) [29] e *Dummy Classifier* (DC) [40]. These algorithms were chosen based on a list of the most frequently used ML algorithms in defect and code smell classification fields [31].

3.4.2 Code Smell Classification with DL Models. DL has been extensively used [20] in code smell detection, primarily due to its ability to analyze large volumes of data and identify complex patterns that traditional ML methods are unable to detect. In our study, the DL algorithms employed are: *Multi-Layer Perceptron* (MLP) [26], *Long Short-Term Memory* (LSTM) [26], *Gated Recurrent Unit* (GRU) [44], and *Autokeras*[3].

3.5 Data Analysis

To evaluate the model responses, we defined three research questions (RQs), as follows:

RQ_1. What is the performance of SLMs in classifying Python code smells? This research question investigates the effectiveness of SLMs in detecting code smells using a baseline prompt engineering technique. To set a reference point for evaluating, we start by establishing a prompt baseline using zero-shot (see Section 3.3.1). We used five open-source SLMs: Deepseek-r1, Qwen2.5-coder, Mistral, Gemma2 and Codllama. The goal is to evaluate whether they can reliably identify smelly code fragments and classify them appropriately without prior fine-tuning. As our ground truth, we used the manual classification labels provided by the PySmell dataset [42] to determine how well SLMs align with expert judgments in real-world codebases.

RQ_2. To what extent do other prompt engineering techniques differ from our baseline for executing the same tasks, in terms of performance? Building on our baseline, we compared other prompt engineering techniques. Among these comparisons, we explore whether SLMs, when guided by more refined prompting strategies, can match or surpass the effectiveness of traditional static analysis tools that rely on AST metrics. For that purpose, we contrast the best-performing SLM (from our baseline) with a well-established metric-based tool, DPY [2], commonly used for code smell detection. This helps clarify both the advantages and limitations of SLM-based approaches when considered as alternatives to conventional metric-based techniques. By combining these dimensions, our analysis provides actionable insights into how different prompt engineering techniques impact the overall performance of SLMs in software analysis tasks.

RQ_3 What is the performance of SLMs in classifying Python code smells compared to traditional ML and DL approaches? This research question investigates whether SLMs can match or surpass the performance of traditional ML and DL approaches in classifying Python code smells. By comparing these paradigms, we aim to understand the trade-offs in accuracy, generalizability, and

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

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

effort required for training and deploying models for automated code quality assessment.

To answer these research questions, we employed scripts developed in Python, responsible for extracting the results stored in the code_smells.db database (Figure 1 – Phase 3) and converting them into CSV files. These files were properly organized to enable the comparison with the data labeled in Phases 2 and 4, and thus quantitative analysis of model performance, using standard evaluation metrics: accuracy, precision, recall, and F1-score. Additionally, the data were organized to facilitate the replication of the study and to support future investigations in the field.

4 Results

We present and discuss the results related to our RQs as follows.

4.1 Baseline Evaluation of Zero-Shot (RQ_1)

An empirical evaluation was conducted using the prompt engineering technique in a zero-shot scenario as the baseline, with the objective of analyzing the capability of SLMs in the automatic classification of code smells in Python.

For the LM classification task, the results presented in Table 2 show that the performance of the evaluated models varied significantly. Among the five open-source SLMs evaluated, Mistral achieved the best performance, with an F1-score of 68%, precision of 83%, and recall of 68%, indicating a high capacity to correctly classify code snippets. The Codellama model also showed competitive performance, with an F1-score of 57%, followed by Qwen2.5-coder with 51%. Conversely, Gemma2 obtained the lowest F1-score (31%), evidencing significant limitations in its effectiveness for this task.

Similarly, in the LPL classification task, the results presented in Table 2 reveal notable differences in performance among the models. Mistral once again stood out, achieving the highest F1-score (59%), with precision of 57% and recall of 65%, indicating consistent capability in identifying methods with excessive parameter lists. Subsequently, Codellama demonstrated competitive performance with an F1-score of 46%, while Deepseek-r1 obtained 48%. On the other hand, Gemma2 exhibited a considerably lower F1-score (10%), suggesting significant limitations in detecting this type of smell.

Overall, the results indicate that some SLMs, especially Mistral, are capable of reliably detecting LM and LPL code smells in Python, even without the need for fine-tuning or additional contextual information.

4.2 Impact of Prompt Engineering Techniques on SLM Performance (RQ_2)

To address this research question, we conducted a comparative analysis of prompt engineering techniques,zero-shot and CoT with and without context. Table 2 shows that performance varied across techniques for both LM and LPL classification.

For the LM classification, Mistral consistently outperformed other models, with F1-score increasing from 68% zero-shot (Baseline) to 87% CoT and 85% CoT with context, evidencing the value of reasoning chains. Context alone (Zero-shot with AST metrics) yielded no gain—remaining at 47%. For the LPL smell, CoT led to no improvements 53% (CoT) and 51% (CoT with context), in comparison to Zero-shot 59% (Baseline) and 55% (with context). Some models, like Gemma2, remained ineffective regardless of technique. These results suggest that the use of CoT enhances classification performance for the LM smell, but remains largely unchanged with LPL, possibly due to the more straightforward syntactic nature of this smell. Additionally, incorporating AST-derived context did not lead to consistent improvements across models.

Although CoT introduced computational overhead due to prompt complexity, it proved useful in explainability, identifying design violations, justifying predictions, and suggesting refactorings, such as method decomposition and parameter reduction to improve cohesion. These interpretative outputs enhance the practical utility of SLMs in software engineering tasks, supporting not only classification but also guiding developers with actionable recommendations for improving code quality.

Finally, when contrasted, the best performance achieved by the Mistral model with DPY (see Table 3). Mistral outperformed DPY in detecting the LPL smell, while DPY outperformed Mistral in detecting the LM smell. However, DPY lacks interpretability and it is not a context—sensitive tool, which are essential factors in choosing between SLM-based and metric-based approaches.

4.3 SLMs versus ML and DL Models (RQ_3)

Table 3 shows the performance results of the SLMs compared to the traditional ML and DL models. When analyzing the LM smell, the DL Autokeras (Autok) and GRU models did not achieve competitive results, presenting an F1-score of only 18% and 32%, respectively. In contrast, the top-3 ML-based models (LDA, SGD, and QDA) showed F1-scores close to 64%. In this case, the SLM Mistral outperformed, achieving an F1-score of 87%.

When we analyzed the LPL smell, we noticed that Autok performed even worse, reaching only 5% in F1-score. In addition, ML models such as ERT, RF, and DT showed very low F1-scores for LPL, remaining close to 15%. In contrast, Mistral achieved an F1-score of 59%, while the AST-based DPY model also delivered consistent results, reaching 53%. The remaining DL models MLP and GRU presented similarly low results, with F1-scores of 4%.

The results demonstrate that in this scenario, the DL and ML models did not exhibit competitive performance in the task of classifying code smells in Python compared to SLMs. Among the approaches evaluated, Mistral and DPY obtained the strongest and most consistent results, especially for LM classification.

Overall, the findings suggest that SLMs, such as Mistral, represent a promising alternative to traditional ML and DL techniques for automated code smell classification. They offer an advantageous balance between predictive performance, inference efficiency, and ease of deployment. Moreover, the potential of prompt-based approaches is highlighted, as they eliminate the need for costly training and hyperparameter tuning processes, making language models particularly attractive for practical applications in code smell classification.

5 Threats to Validity

This analysis focused solely on Python, which may limit result generalizability. While Python is widely adopted, further studies are needed to verify if the observed patterns extend to other languages. Additionally, the study addressed only two code smells—LM

Table 2: Performance comparison of SLMs using different prompt engineering techniques for code smell classification

Class	Model	Zero-shot (Baseline)			Zero-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
LM	Deepseek-r1	0.47	0.48	0.47	0.49	0.49	0.49	0.42	0.45	0.36	0.21	0.38	0.23
	Qwen2.5-coder	0.51	0.51	0.51	0.53	0.53	0.52	0.53	0.53	0.52	0.35	0.45	0.28
	Gemma2	0.41	0.47	0.31	0.39	0.44	0.31	0.16	0.43	0.24	0.15	0.37	0.21
	Mistral	**0.83**	**0.68**	**0.68**	**0.83**	0.54	0.47	**0.90**	**0.85**	**0.87**	**0.86**	**0.84**	**0.85**
	Codellama	0.58	0.57	0.57	0.64	**0.64**	**0.64**	0.48	0.48	0.40	0.52	0.51	0.43
LPL	Deepseek-r1	0.50	0.48	0.48	0.50	0.50	0.46	0.50	0.46	0.19	0.50	0.51	0.18
	Qwen2.5-coder	0.50	0.51	0.41	0.50	0.52	0.39	0.50	0.51	0.15	0.51	0.51	0.10
	Gemma2	0.51	0.54	0.10	0.51	0.54	0.10	0.51	0.50	0.02	0.51	0.50	0.02
	Mistral	**0.57**	**0.65**	**0.59**	**0.67**	0.53	**0.55**	**0.53**	**0.68**	**0.53**	**0.52**	**0.66**	**0.51**
	Codellama	0.50	0.54	0.46	0.50	**0.55**	0.42	0.51	0.53	0.10	0.51	0.54	0.18

Table 3: Comparison of SLM (Mistral), DPY, ML, and DL models per code smell class, including execution time

Model	Class	Accuracy	Precision	Recall	F1-score	Exec. Time (s)
Best-SLM (Mistral)						
Mistral	LM	0.89	**0.90**	**0.85**	**0.87**	1.992
	LPL	**0.95**	0.57	0.65	0.59	25.670
AST-based (DPY)						
DPY	LM	**0.96**	**0.98**	**0.96**	**0.97**	5
	LPL	**0.96**	0.53	0.54	0.53	64
Top-3 ML Models						
LDA	LM	**0.29**	0.59	0.71	**0.64**	0.655
SGD		0.07	**0.62**	0.65	**0.64**	0.806
QDA		0.10	0.52	**0.76**	0.62	0.648
ERT	LPL	**0.51**	**0.12**	**0.22**	**0.15**	1.429
RF		**0.51**	0.11	0.20	0.14	1.993
DT		**0.51**	0.10	0.15	0.12	0.711
Top-3 DL Models						
AutoK	LM	0.81	0.11	**0.56**	0.18	0.146
MLP		**0.90**	**0.75**	0.05	0.10	0.219
GRU		**0.90**	0.57	0.22	**0.32**	0.219
AutoK	LPL	0.81	0.04	**0.09**	**0.05**	0.146
MLP		**0.90**	**1.00**	0.02	0.04	0.219
GRU		**0.90**	**1.00**	0.02	0.04	0.219

and LPL—potentially narrowing the scope. Future work should explore other code smells to assess model performance across varying abstraction levels and broaden applicability.

This study exclusively adopted two prompting strategies—zero-shot and CoT—to guide the models in classifying the provided code samples. Although these approaches are widely recognized in the literature, the restricted adoption of techniques may not fully reflect the inferential capabilities of SLMs. Future research could consider a broader range of prompting strategies to provide a more comprehensive understanding of model behavior in the task of code smell classification. Moreover, the SLMs evaluated in this work—despite their advantages in terms of computational efficiency and deployment feasibility—are inherently constrained by their reduced parameter count. It may affect the full comprehension of larger files and complex program structures. In future work, more sophisticated techniques (*e.g.*, combining SLMs with AST-derived features) could help overcome these limitations.

6 Conclusion and Future Plans

In this work, we have presented a systematic evaluation of SLMs for the detection of two pervasive Python code smells—Long Method (LM) and Long Parameter List (LPL)—and compared their performance against traditional AST-based analysis, classical ML and DL approaches. Our experiments demonstrate that SLMs showed a good performance when analyzing and detecting the code smells LM and LPL, and may serve as a foundation for building lightweight, intelligent code quality solutions in modern development workflows. Despite their promising performance, there remains room for improvement—especially in refining prompt engineering techniques and exploring model-specific adaptations.

In the detection of smells using the zero-shot as the base prompt, Mistral obtained the best performance for both LM and LPL, standing out for LM with an accuracy value of 83%, Recall 68% and F1-Score 68%. For LM, CoT without and with context prompts outperformed the Zero-shot (Baseline) for Mistral with an F1-score of 87% and 85%, respectively.

When comparing the ML and DL models with SLM, we noticed that AutoK did not perform very well for either LM or LPL. Although ML did not perform very well for LM, for LPL it also remained below Mistral.

As next steps, we will evaluate the interpretability of SLM and learning-generated explanations in cases where their predictions diverge from ground truth labels. We want to better understand whether the reasoning provided by different SLMs and ML/DL models (e.g., via chain-of-thought prompts and SHAPE) may help developers understand the source of error or ambiguity. The aim is to evaluate the clarity, relevance, and reliability of these rationales, particularly in borderline or complex cases where traditional tools (*i.e.,* DPY) may struggle and SLMs may misclassify code fragments. Understanding these explanations is essential for establishing trust in SLM-assisted software quality tasks.

ARTIFACT AVAILABILITY

We have made all research artifacts supporting this study's findings publicly available at https://zenodo.org/records/17074239. The corresponding GitHub repository is also publicly available at https://github.com/aisepucrio/lm4smells-core [34].

ACKNOWLEDGMENTS

This research was partially funded by the Brazilian funding agencies CAPES/COFECUB (Grant 88881.879016/2023-01 and Ma1036/24), FAPESP (Grant 2023/00811-0). We also acknowledge the Brazilian company Stone[4] for their financial support.

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

REFERENCES

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

[2] Aryan Boloori and Tushar Sharma. 2025. DPy: Code Smells Detection Tool for Python. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 826–830.

[3] 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.

[4] Opher Etzion and Peter Niblett. 2010. *Event Processing in Action*. Manning Publications. https://www.manning.com/books/event-processing-in-action

[5] 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.

[6] 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 '16)*. Association for Computing Machinery, Article 18, 12 pages. https://doi.org/10.1145/2915970.2915984

[7] Python Software Foundation. 2024. AST — Abstract Syntax Trees. https://docs.python.org/3/library/ast.html#module-ast Accessed on: December 24, 2024.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 2012. *Refactoring: Improving the Design of Existing Code*. Pearson Education. https://books.google.com.br/books?id=HmrDHwgkbPsC

[9] 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.

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

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

[12] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (Nov. 2023), 41 pages. https://doi.org/10.1145/3617175

[13] 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* 20, 4 (2024), 307–316.

[14] Aurélien Géron. 2019. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Incorporated.

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

[16] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional. https://www.amazon.com/dp/0321200683

[17] 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 *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 840–849.

[18] 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.

[19] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. arXiv:2205.11916 [cs.CL] https://arxiv.org/abs/2205.11916

[20] 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.

[21] R.C. Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. https://books.google.com.br/books?id=hjEFCAAAQBAJ

[22] R.C. Martin. 2020. Solid relevance. https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html Accessed on: July 01, 2025.

[23] Vinícius Martins, Pedro Lopes Verardo Ramos, Breno Braga Neves, Maria Vitoria Lima, Johny Arriel, João Victor Godinho, Joanne Ribeiro, Alessandro Garcia, and Juliana Alves Pereira. 2024. Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis. In *XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 302–312. https://doi.org/10.5753/sbes.2024.3431

[24] M. Masse. 2011. *REST API Design Rulebook*. O'Reilly Media. https://books.google.com.br/books?id=4lZcsRwXo6MC

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

[26] Himesh Nanadani, Mootez Saad, and Tushar Sharma. 2023. Calibrating Deep Learning-based Code Smell Detection using Human Feedback. In *23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 37–48.

[27] Ollama. 2025. Ollama. https://ollama.com/ Accessed on: July 2, 2025.

[28] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 7. https://doi.org/10.1186/s40411-017-0041-1

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

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

[31] 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: 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings*. Springer-Verlag, 282–305. https://doi.org/10.1007/978-3-031-30826-0_16

[32] Amazon Web Services. 2024. Anti-Corruption Layer (ACL). https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/acl.html Accessed on: July 24, 2025.

[33] 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 (ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 400–406. https://doi.org/10.1145/3674805.3690742

[34] Igor Soares de Oliveira, Joanne Carneiro, Jessica Ribas, and Juliana Alves Pereira. 2025. Code Smell Classification in Python: Are Small Language Models Up to the Task? https://github.com/aisepucrio/lm4smells-core Accessed on: July 15, 2025.

[35] 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

[36] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 329–331.

[37] Santiago A. Vidal, Claudia Marcos, and J. Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engg.* 23, 3 (Sept. 2016), 501–532. https://doi.org/10.1007/s10515-014-0175-x

[38] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, Qi He, Yao Ma, Ming Huang, and Suhang Wang. 2024. A Comprehensive Survey of Small Language Models in the Era of Large Language Models: Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness. arXiv:2411.03350 [cs.CL] https://arxiv.org/abs/2411.03350

[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

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

[41] 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 '24)*. Association for Computing Machinery, 1345–1357. https://doi.org/10.1145/3691620.3695508

[42] Zhifei Chen; Lin Chen; Wanwangying Ma; Baowen Xu. 2025. Pysmell. https://github.com/chenzhifei731/Pysmell Accessed on: April 28, 2025.

[43] 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.

[44] Dongwen Zhang, Shuai Song, Yang Zhang, Haiyang Liu, and Gaojie Shen. 2023. Code Smell Detection Research Based on Pre-training and Stacking Models. *Latin America Transactions* 22, 1 (2023), 22–30.

[45] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. 2022. Code smells for machine learning applications. In *Proceedings of the 1st international conference on AI engineering: software engineering for AI*. IEEE, 217–228.