Attentionsmelling: Using Large Language Models to Identify Code Smells

Anderson Gomes
State University of Ceará
Fortaleza, Ceará, Brazil
anderson.martins@aluno.uece.br

Denis Sousa
State University of Ceará
Fortaleza, Ceará, Brazil
denis.sousa@aluno.uece.br

Paulo Maia
State University of Ceará
Fortaleza, Ceará, Brazil
pauloh.maia@uece.br

Matheus Paixao
State University of Ceará
Fortaleza, Ceará, Brazil
matheus.paixao@uece.br

ABSTRACT

Large Language Models (LLMs) are becoming essential tools in software engineering, automating tasks like code generation, unit testing, and code review. However, their potential to identify code smells, indicators of poor code quality, remains underexplored. This study evaluates GPT-4's effectiveness in identifying three common code smells (Long Method, God Class, and Feature Envy) across four experimental setups ranging from using only source code and code smell definitions to leveraging additional context, metrics, and hyperparameter optimization. Our analysis revealed remarkable improvements across all metrics when improving the prompt with more information, with overall performance increasing by 64% in ROC Curve and 56% in F1-score. These results emphasize the impact of incorporating metrics and hyperparameter tuning into LLM prompts, enabling significant advancements in automated software quality assessment. This may lead to better coding practices, particularly related to identifying code smells.

KEYWORDS

Large Language Models, Code Smell Detection, Software Quality, Prompt Engineering, Software Maintenance, Neural Networks

1 Introduction

Large Language Models (LLMs) are advanced machine learning models designed to understand and generate human language by analyzing vast datasets of text [4]. These models, such as GPT-4 [1], use complex neural network architectures, typically based on transformers, to predict and generate text by processing word sequences and learning contextual patterns.

LLMs have shown considerable potential for various software engineering activities, enhancing developer productivity. For example, LLMs can assist in code generation [6], unit test generation [19], code documentation [11], code review and bug identification [3], support in the development of systems with multiple programming languages [2] among others. These activities represent an important step, allowing developers to focus more on complex problem-solving while relying on AI for routine or error-prone tasks.

LLMs are being widely used to identify, classify and refactor code smells [10, 14, 18, 20], which are indicators of potential issues that suggest code weaknesses that may lead to problems of readability and extensibility [7]. Traditionally, code smell identification relies on static analysis tools like SonarQube [21] and Checkstyle [5], which operate based on predefined rule sets and lack flexibility to capture complex, contextual patterns. In contrast, LLMs offer the ability to understand context and are also able to capture the developer preference, potentially allowing the identification of more subtle code smells.

Currently, we have found only two studies addressing the use of LLMs for code smells identification [14, 20]. The study conducted by Lucas et al. [14] evaluated the ability of three LLMs (ChatGPT-4, Mistral Large, and Gemini Advanced) to identify test smells. The models were evaluated on 30 types of test smells across codebases written in seven different programming languages. The authors focused exclusively on test smells rather than code smells and applied only the simple zero-shot prompt approach [15, 24]. Silva et al. [20] utilized ChatGPT with two prompting strategies (generic and mentions) to identify four code smells in Java projects. Although the prompts in the latter study are slightly more advanced, they only provide information about the code smells that the LLM should detect. They do not offer precise descriptions of each code smell or additional context.

In this study, we aim to systematically evaluate how effectively GPT-4o can identify code smells under different configurations. Our investigation is structured into four experiments, each assessing a specific factor influencing LLM performance: (EXP1) the effectiveness of mention-based prompts as a baseline, (EXP2) the impact of explicit code smell descriptions, (EXP3) the effect of incorporating structured metric data, and (EXP4) the influence of hyperparameter tuning. This structured approach provides a deeper understanding of the potential and limitations of LLMs for automated software quality assessment.

To enhance the GPT-4o's performance, we used specialized prompts explicitly designed for each type of code smell. These prompts have precise definitions for each code smell, ensuring clarity and specificity. Additionally, to improve their effectiveness, we incorporated metrics linked to the code snippets and tuned the model's hyperparameters. This provided additional context and guided the model towards more deterministic behavior.

Furthermore, we created a new code smell oracle [8] by adapting the work of Reis et al. [17], which proposes the CrowdSmelling oracle containing code smells written in Java. The original dataset consists of classifications performed by different teams of students, who identified code smells either manually or with assistance from

the JDeodorant plugin. To mitigate potential biases introduced by human subjectivity and tool limitations, we applied a majority voting approach to determine the final classification of each smell instance. Additionally, we validated our results against independent subsets of the original dataset to assess the reliability of our new oracle. The final dataset contains only instances with high-confidence classifications, reducing uncertainty in our evaluation.

Unlike the CrowdSmelling oracle, which includes multiple evaluations for the same smell, the new oracle presents a single definitive evaluation of whether they represent a true code smell. Additionally, we included a confidence level (high, medium, and low) for classifying the code smells, considering that certain smells had very close numbers of true and false classifications. This scenario demonstrates that evaluating whether a piece of code constitutes a smell was challenging, even for humans.

The remainder of this work is divided as follows: Section 2 presents the study design that guided our evaluation, while Section 3 details the main results found. Section 4 discusses not only the insights and implications of this work, but also provides further research directions. The threats to validity are addressed in Section 5 and the main related work is compared in Section 6. Finally, Section 7 draws the conclusions and future work.

2 Study Design

2.1 New Oracle

In this study, we used the Crowdsmelling oracle produced by Reis et al. [17], which documents code smells in Java projects. The oracle was created using the Crowdsmelling approach, which leverages the collective wisdom (or "crowd") of various students who collaborate to identify and validate the presence of code smells in Java projects. Over three consecutive years (2018, 2019, and 2020), 103 student teams, with an average of 3 members each, classified the presence of three types of code smells: Long Method, God Class, and Feature Envy.

The classification process was both manual and automated, aided by the tool JDeodorant[1], which identifies code smells and collects code metrics. The teams analyzed 3 projects: jasml-0.10[2], jgrapht-0.8.1[3], and jfreechart-1.0.13[4] and decided to accept (true positives) or reject (false positives) the tool's suggestions or add additional manual identifications.

The original dataset contains multiple classifications for the same smell due to repeated evaluations of the same methods or classes by different teams. To consolidate this information, we applied a structured aggregation process, grouping annotations that refer to the same code entity. This ensures that our new oracle remains aligned with the original dataset while reducing duplicate instances and conflicting labels. This grouping process also allows us to maintain traceability between the final dataset and the source evaluations, ensuring that the insights drawn from our study remain representative of real-world software quality assessments.

Nonetheless, the CrowdSmelling oracle presents certain limitations, which required adaptations to its data for the purposes of this study. First, in terms of dataset composition, the oracle contains a total of 1943 smell records, with only 30 belonging to the projects *jgrapht-0.8.1* and *jfreechart-1.0.13*, making them a minority. Consequently, only the *jasml-0.10* project was included in the new oracle to maintain dataset consistency and focus.

Second, regarding the structure of the data, the original oracle provides smell classifications at both the method and class levels, but lacks full code context. To address this, the identified smells were grouped by their method or class signatures along with their package names. This grouping preserved the structural relationships within the code and allowed for the consolidation of multiple annotations for the same entity. The final classification was determined by a majority vote across annotations. This approach reduced noise and inconsistencies arising from individual assessments and ensured that each instance corresponded to a distinct, well-contextualized code smell occurrence. After this grouping process, the number of code smells was reduced to 135.

Finally, to capture the complete code, a challenge arose from the presence of different signatures referring to the same code snippet. In order to eliminate duplicates and retrieve the full code corresponding to each signature, GPT-4 was employed. This treatment further refined the dataset, resulting in a final set of 80 code smell records.

These adaptations resulted in a new oracle that contains 42 Long Methods (20 true and 22 false), 9 God Classes (7 true and 2 false), and 29 Feature Envy (21 true and 8 false) instances, summing up 80 code smells exemplars.

2.2 Evaluation Metrics

To evaluate LLM's performance in identifying code smells, we used common metrics from machine learning-based systems [12, 27, 29, 30] and LLMs [6, 9, 14, 16, 20]: (1) **Accuracy**, which measures the proportion of correct predictions relative to total predictions; (2) **F1-Score**, the harmonic mean of the metrics precision and recall, thus providing a balanced representation of both metrics in a single measure. Precision indicates the proportion of code smells identified by the model that are true positives, while Recall measures the proportion of actual code smells in the oracle that were correctly identified by the model; and (3) **ROC Curve (AUC)**, which represents the area under the Receiver Operating Characteristic curve, showing the trade-off between true positive and false positive rates across various thresholds.

2.3 Research Question and Experimental Steps

In this study, we aim to evaluate the effectiveness of a state-of-the-art Large Language Model (LLM), GPT-4o, in identifying and classifying code smells across different configurations.

To guide our investigation, we define the following research question:

RQ: *How effectively can a Large Language Model (LLM) classify code smells under different prompt configurations, contextual enhancements, and optimization strategies?*

[1]https://users.encs.concordia.ca/ nikolaos/jdeodorant/
[2]https://jasml.sourceforge.net/
[3]https://jgrapht.org/
[4]https://www.jfree.org/

Therefore, our study investigates how GPT-4o copes with the dual task of *identifying* whether a smell exists and then *classifying* it into a concrete smell category.[5]

In other words, we follow the usual machine-learning convention whereby *classification subsumes identification*: a tool that "classifies a code smell" must first detect (i.e., identify) its presence and only then allocate the instance to the correct smell category. Thus, whenever the remainder of this paper says that GPT-4o *classifies* a smell, the identification step is implicitly included.

To systematically address this question, we designed a series of experiments focused on a specific factor influencing the ability of GPT-4o to detect code smells. Each step builds upon the previous one, allowing for a progressive refinement of GPT-4o's classification methodology:

2.3.1 Experiment 1 (EXP1): Baseline Performance with Mention-Based Prompts. **Objective:** Establish a baseline by evaluating GPT-4o's ability to classify code smells using mention-based prompts. **Description:** Following the approach proposed by Silva et al. [20], we assess the model's performance when provided only with prompts that *mention* the names of code smells, without explicit definitions. The goal is to determine whether GPT-4o can infer and correctly identify code smells based solely on their names.

2.3.2 Experiment 2 (EXP2): Performance Improvement with Specialized Prompts. **Objective:** Investigate the impact of explicitly describing code smells in prompts on classification accuracy. **Description:** Instead of relying on mention-based prompts, we introduce *specialized prompts* containing structured definitions of each code smell, extracted from Reis et al. [17]. These refined prompts aim to provide a more informative basis for GPT-4o's classification process.

2.3.3 Experiment 3 (EXP3): Effect of Incorporating Code Metrics. **Objective:** Analyze whether integrating quantitative code metrics into the specialized prompts enhances classification accuracy. **Description:** Since LLMs may lack an inherent understanding of software quality metrics, we enhance the prompts by incorporating structured metric data sourced from the *CrowdSmelling oracle* [17]. This additional context provides GPT-4o with quantitative insights into structural and behavioral aspects of the code. Long Method and Feature Envy are evaluated using 82 metrics each, while God Class is evaluated with 61 metrics.

2.3.4 Experiment 4 (EXP4): Impact of Hyperparameter Tuning. **Objective:** Assess the effect of fine-tuning GPT-4o's hyperparameters on classification consistency and reliability. **Description:** We build upon Experiment 3 by adjusting key hyperparameters, such as *temperature*, *top-p*, *frequency penalty*, and *presence penalty*, to optimize response determinism and precision. Adjusting these parameters can significantly influence LLM performance [22, 23, 28]. The tuning process follows a more deterministic configuration, with the following parameter values: *temperature*=0.3, *top-p*=0.9, *frequency penalty*=0.5, and *presence penalty*=0.2[6].

[5]The replication package containing all prompts and scripts is available at https://anonymous.4open.science/attentionsmelling.

[6]These values correspond to OpenAI's recommended defaults for code-analysis tasks and were confirmed in a five-case pilot sweep that maximised F1 score (details in our replication package).

These four experimental steps progressively refine GPT-4o's classification methodology, enabling a comprehensive assessment of how different prompt configurations, contextual enhancements, and optimization strategies influence its ability to identify code smells effectively.

2.4 Specialized Prompts

For each type of code smell investigated (Long Method, Feature Envy and God Class), a specialized prompt containing its description was designed for EXP2, EXP3, and EXP4. The code smell definitions used in the prompts were extracted from the study by Reis et al. [17]. Figures 1, 2, and 3 present the specialized prompts used to identify the Long Method, God Class, and Feature Envy code smells, respectively. In each prompt, was omitted the sections related to the LLM persona, instructions and response formatting.

Long Method Prompt

```
«PERSONA»
A "Long Method" is characterized by:
- Being excessively long.
- Performing multiple tasks or having multiple
responsibilities.
- Having high complexity due to nested structures
(loops, conditionals).
- Being hard to understand, maintain, or extend.
Ideally, a method should focus on a single
responsibility.
**Instructions:** «INSTRUCTIONS»
**Your response should be in the following JSON
format:** «JSON_FORMAT»
```

Figure 1: Prompt Specialized is designed for LLMs to identify Long Method.

The Figure 4 presents the persona section used by GPT-4o for it to perform code smells identification, while Figure 5 presents the behavior instruction and response formatting sections of the prompt. We highlighted the keyword "CODE-SMELL-LEVEL" to specify whether the code smell applies at the "class" or "method" level, and "CODE-SMELL-NAME" to indicate the smell name.

In this study, the mentions prompts proposed by Silva et al. [20] were also utilized (EXP1). These prompts contain a list of code smell names that the LLM is tasked with identifying. If the model identify any of the smells listed, it is also required to specify which code smell was identified. An example response from the model could be: "YES, I found bad smells. The bad smells are: 1. Long Method."

3 Results

This section presents the performance of GPT-4o in identifying code smells using F1-Score, ROC Curve (AUC), and Accuracy as the evaluation metrics. The analysis is structured to answer the four experiments concerning the identification of the code smells Long Method, God Class, and Feature Envy. In addition, we included the item "All Code Smells" that represents the application of the metrics to the whole dataset without discriminating the type of code smell.

God Class Prompt

```
«PERSONA»
A "God Class" is characterized by:
- Accumulating too many responsibilities, often
having a large number of methods and attributes.
- Managing multiple, unrelated functionalities within
a single class.
- Poor cohesion and high dependency on other classes,
making it difficult to maintain and extend.
- Violating the Single Responsibility Principle
(SRP).
Ideally, a class should focus on a single,
well-defined responsibility.
**Instructions:** «INSTRUCTIONS»
**Your response should be in the following JSON
format:** «JSON_FORMAT»
```

Figure 2: Prompt Specialized is designed for LLMs to identify God Class.

Feature Envy Prompt

```
«PERSONA»
A "Feature Envy" code smell is characterized by:
- A method that accesses and relies heavily on the
data and
functions of another class rather than its own.
- Displaying more interest in the members of other
classes, which suggests that it might be better
placed in the other class.
- Performing operations on another class's attributes
or methods, indicating a possible misplacement of
the method.
Ideally, a method should focus on the data and
functions within its own class.
**Instructions:** «INSTRUCTIONS»
**Your response should be in the following JSON
format:** «JSON_FORMAT»
```

Figure 3: Prompt Specialized is designed for LLMs to identify Feature Envy.

3.1 Performance Metrics Overview

The results are summarized in Table 1 and Table 2, which presents the F1-Score, AUC, and Accuracy for each code smell across the four experiments (EXP1 to EXP4). These metrics collectively reflect the model's effectiveness in identifying code smells as the research progressed.

For **all code smells**, the results indicate a steady improvement from EXP1 to EXP4. The AUC increases from 0.50 in EXP1 to 0.82 in EXP4, while Accuracy rises from 0.51 to 0.75 over the same period. Similarly, the F1-Score improves from 0.50 in EXP1 to 0.78 in EXP4, demonstrating the model's growing capability in detecting code smells across the board.

Persona Prompt

```
You are an expert Java software engineer
specializing in code smells, particularly the
CODE-SMELL-NAME code smell.
When provided with a Java CODE-SMELL-LEVEL, your
task is to analyze it to determine if it exhibits
the CODE-SMELL-NAME code smell.
```

Figure 4: Section Prompt used to specify the persona of the LLM.

Instructions and Response Formatting Prompt

```
**Instructions:**
1. **Classification**: Determine if the
CODE-SMELL-LEVEL is a case of CODE-SMELL-NAME.
Use "True" if it is, otherwise "False".
2. Classify the CODE-SMELL-LEVEL as
CODE-SMELL-NAME only if you are confident in your
analysis. That is, be quite strict and
restrictive when identifying this Code Smell.
3. The CODE-SMELL-LEVEL's body is delimited by
'***SOURCE CODE: BEGIN***' and '***SOURCE CODE:
END***'
4. Keep in mind the objective metrics provided in
your evaluation.
5. The metrics are delimited by '***METRICS:
BEGIN***' and '***METRICS: END***'
**Your response should be in the following JSON
format:**
"'json
{
  "classification": "True" or "False",
}
```

Figure 5: Section Prompt used to specify Instructions and Response Formatting.

In the case of **Feature Envy**, the results show a relatively stable AUC across the experiments, starting at 0.50 in EXP1, peaking at 0.61 in EXP2, and settling at 0.54 in EXP3 and EXP4. However, Accuracy improves significantly, increasing from 0.29 in EXP1 to 0.71 in EXP4. The F1-Score also exhibits notable growth, beginning at 0.13 in EXP1 and reaching 0.65 in EXP4.

For **God Class**, the metrics reveal a clear upward trend. The AUC improves from 0.61 in EXP1 to a perfect score of 1.00 in both EXP3 and EXP4. Furthermore, both Accuracy and F1-Score reach 1.00 in these later experiments, indicating flawless classification performance.

Regarding **Long Method**, the performance metrics peak during EXP2. The AUC increases from 0.63 in EXP1 to 0.78 in EXP2, followed by a decline in subsequent experiments. Accuracy follows a similar trajectory.

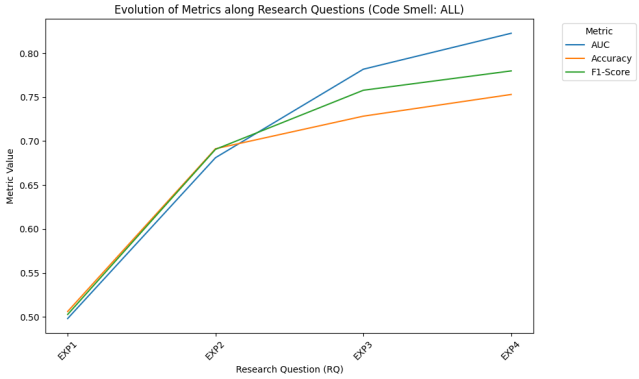

Figure 6: Evolution of Metrics Along Experiments (ALL Code Smells)

The evolution of metrics across experiments is illustrated in Figure 6, providing a visual representation of these trends.

3.2 Results by Experiment

3.2.1 EXP1 - Baseline Performance with Mention-Based Prompts Proposed by Silva et al. [20]. In the first experiment (EXP1), the model was evaluated using the mention-based prompts originally proposed by Silva et al. [20]. For **all code smells**, the model demonstrated moderate performance, with an F1-Score of 0.50, an AUC of 0.50, and an Accuracy of 0.51. These results indicate that while the mention prompt establishes a reasonable baseline for smell detection, it leaves significant room for improvement when identifying smells collectively.

When analyzing **Long Method**, the model achieved comparatively better results, with an F1-Score of 0.56, an AUC of 0.63, and an Accuracy of 0.61. These outcomes suggest that the mention prompt is somewhat effective in detecting Long Method instances, although there are notable limitations in both precision and recall.

For **God Class**, the model performed well, achieving an F1-Score of 0.69, an AUC of 0.61, and an Accuracy of 0.67. These results highlight the model's ability to identify God Class instances with reasonable accuracy and reliability, even with a generalized prompt.

Conversely, **Feature Envy** yielded the weakest performance, with an F1-Score of only 0.13, an AUC of 0.50, and an Accuracy of 0.29. These metrics suggest substantial difficulty in detecting this smell, likely due to the mention prompt's inability to effectively capture the context-sensitive characteristics that define Feature Envy.

3.2.2 EXP2 - Performance Improvement with Specialized Prompts. The second experiment demonstrated the positive effects of using specialized prompts, with performance improvements observed across all code smells. For **all code smells**, the F1-Score increased from 0.50 to 0.69, the AUC rose from 0.50 to 0.68, and Accuracy improved from 0.51 to 0.69. These results reflect the overall enhancement in classification performance brought about by tailoring prompts to each specific smell.

In the case of **Long Method**, the model achieved substantial gains. The F1-Score improved from 0.56 to 0.76, the AUC increased from 0.63 to 0.78, and Accuracy rose from 0.61 to 0.77. These improvements indicate a higher level of reliability in identifying Long Method instances, with a notable reduction in misclassifications due to more targeted prompt structures.

For **God Class**, while the metrics in EXP1 were already strong with an F1-Score of 0.69, an AUC of 0.61, and an Accuracy of 0.67—specialized prompts in EXP2 further enhanced the model's effectiveness. The F1-Score rose to 0.87, the AUC increased to 0.75, and Accuracy improved to 0.89. These results underscore the benefits of incorporating structural cues into the prompt design for identifying this particular smell.

Finally, **Feature Envy** showed noticeable progress, although it remained the most challenging smell to detect. The F1-Score increased significantly from 0.13 to 0.50, Accuracy improved from 0.29 to 0.50, and AUC rose from 0.50 to 0.61. Despite these advancements, the results indicate that the model still struggles to maintain a balanced trade-off between precision and recall for Feature Envy, likely due to its context-sensitive nature.

Overall, these findings highlight the efficacy of specialized prompt engineering in improving smell-specific performance and reducing classification errors across the board.

3.2.3 EXP3 - Effect of Incorporating Code Metrics. In the third experiment (EXP3), the integration of code metrics into the prompt design led to notable shifts in model performance across all code smells. For **all code smells**, the results improved significantly, with the F1-Score increasing from 0.69 to 0.76, the AUC rising from 0.68 to 0.82, and Accuracy improving from 0.73 to 0.75. These enhancements underscore the value of incorporating metrics into the prompts, particularly due to the perfect classification of God Class and the substantial improvement observed for Feature Envy.

For **Long Method**, however, performance declined compared to EXP2. The F1-Score dropped from 0.76 to 0.68, Accuracy decreased from 0.77 to 0.70, and the AUC fell from 0.78 to 0.72. These results suggest that the addition of metrics may have introduced noise or ambiguity, potentially leading to an increase in false positives or confusion around threshold-based indicators such as method length. While the performance remained moderately effective, these findings highlight the need for careful metric selection and interpretation for Long Method detection.

In contrast, **God Class** achieved perfect performance across all metrics, with an F1-Score, AUC, and Accuracy of 1.00. This outcome indicates that the inclusion of structural metrics—particularly those related to cohesion and dependency—provided the necessary context for consistent and accurate identification of this smell.

Regarding **Feature Envy**, the results demonstrated a marked improvement, with the F1-Score increasing from 0.50 to 0.65 and Accuracy rising from 0.50 to 0.71. Nonetheless, the AUC experienced a slight decline from 0.61 to 0.55, suggesting ongoing challenges in balancing precision and recall. These results reveal that while the integration of metrics strengthens the model's ability to detect Feature Envy, further refinement is needed to reduce misclassifications and improve predictive stability.

Overall, EXP3 illustrates that the incorporation of code metrics can enhance the model's understanding and performance, especially for structurally defined smells like God Class, but also brings new

Table 1: Detailed Performance Metrics for Code Smell Identification

Code Smell	Metric	EXP1 (Mentions Prompts)	EXP2 (Specialized Prompts)	EXP3 (Specialized Prompts + Metrics)	EXP4 (Specialized Prompts + Metrics + Tuning)
All	F1-Score	0.502799	0.690679	0.757878	0.779890
	AUC	0.498157	0.681164	0.781818	0.822727
	Accuracy	0.506173	0.691358	0.728395	0.753086
Long Method	F1-Score	0.556054	0.763015	0.681267	0.709447
	AUC	0.630435	0.782609	0.717391	0.739130
	Accuracy	0.613636	0.772727	0.704545	0.727273
God Class	F1-Score	0.687179	0.874074	1.000000	1.000000
	AUC	0.607143	0.750000	1.000000	1.000000
	Accuracy	0.666667	0.888889	1.000000	1.000000
Feature Envy	F1-Score	0.126984	0.500000	0.623377	0.647205
	AUC	0.500000	0.612500	0.512500	0.537500
	Accuracy	0.285714	0.500000	0.678571	0.714286

Table 2: Best Performance Metrics for Code Smell Identification

Code Smell	Metric	Best EXP	Best Value
All	AUC	EXP4	0.822727
All	Accuracy	EXP4	0.753086
All	F1-Score	EXP4	0.779890
Feature Envy	AUC	EXP2	0.612500
Feature Envy	Accuracy	EXP4	0.714286
Feature Envy	F1-Score	EXP4	0.647205
God Class	AUC	EXP3	1.000000
God Class	Accuracy	EXP3	1.000000
God Class	F1-Score	EXP3	1.000000
Long Method	AUC	EXP2	0.782609
Long Method	Accuracy	EXP2	0.772727
Long Method	F1-Score	EXP2	0.763015

challenges that must be addressed for more context-dependent smells like Long Method and Feature Envy.

3.2.4 EXP4 - Impact of Hyperparameter Tuning. In the fourth experiment (EXP4), the application of hyperparameter tuning led to marginal improvements in model performance. For **all code smells**, the AUC increased from 0.78 to 0.82, Accuracy improved slightly from 0.73 to 0.75, and the F1-Score rose from 0.76 to 0.78. These results indicate that while hyperparameter tuning contributed to refining the model, its overall impact on classification performance was limited, especially compared to earlier interventions like prompt specialization and metric integration.

For **Long Method**, performance showed modest gains. The F1-Score increased from 0.68 to 0.71, the AUC rose from 0.72 to 0.74, and Accuracy improved from 0.71 to 0.73. These changes suggest that tuning hyperparameters slightly enhanced the model's ability to detect Long Method instances, although the improvements remained relatively modest in scale.

God Class continued to achieve perfect performance across all metrics, maintaining an F1-Score, AUC, and Accuracy of 1.00. This consistency reaffirms that hyperparameter tuning did not yield any additional gains beyond those already achieved through the incorporation of structural metrics in EXP3.

Regarding **Feature Envy**, performance remained virtually unchanged, with Accuracy holding steady at 0.71, F1-Score at 0.65, and AUC slightly increasing to 0.58. These results indicate that hyperparameter tuning was insufficient to address the lingering challenges in balancing precision and recall for this smell. The model continued to struggle with distinguishing legitimate inter-class interactions from true instances of Feature Envy.

Overall, the findings from EXP4 suggest that while hyperparameter tuning can provide incremental improvements, its effectiveness is contingent on the foundations laid by earlier stages of prompt and metric optimization.

3.3 Confidence Analysis for Misclassifications in Code Smell Detection

Table 3 presents an in-depth analysis of the confidence levels associated with the misclassifications identified during the evaluation of EXP4, which achieved the highest overall performance in the study. This analysis focuses exclusively on instances with more than one vote, as these provide more reliable insights into the model's classification behavior.

The table includes several key columns. The **Votes Count** indicates the total number of annotations—both positive and negative—used to evaluate a given instance. **Negative Votes** and **Positive Votes** represent the number of responses suggesting the absence or presence of a code smell, respectively. The **Ground Truth** reflects the majority consensus classification based on these votes, with a value of `true` or `false` indicating the presence or absence of the smell. This serves as the reference against which the model's **Predicted Outcome** is compared.

The **Ground Truth Confidence** is a numerical value ranging from 0 to 1 representing the model's confidence in the ground truth label. The confidence level (CL) for a specific code smell is calculated using the following Equation 1:

$$CL = \frac{\max(P, N)}{T} \tag{1}$$

P and N represent the number of positive and negative classifications respectively, and T is the total number of evaluations made for that specific code smell. The maximum value between P and N is divided by T to determine the CL.

Finally, the **Confidence Level**: The confidence level categorized as `HIGH`, `MEDIUM`, or `LOW`, based on the ground truth confidence value as follows:

$CL > 0.85$ is classified as high confidence,
$0.70 \leq CL \leq 0.85$ as medium confidence, and
$CL < 0.70$ as low confidence.

Table 3: Confidence Analysis for Code Smell Detection

Index	Code Smell	Votes Count	Negative Votes	Positive Votes	Ground Truth	Predicted Outcome	Ground Truth Confidence	Confidence Level
1	Feature Envy	2	0	2	true	false	1.00	HIGH
2	Long Method	44	42	2	false	true	0.95	HIGH
3	Long Method	38	36	2	false	true	0.95	HIGH
4	Long Method	45	42	3	false	true	0.93	HIGH
5	Long Method	16	13	3	false	true	0.81	MEDIUM
6	Long Method	63	51	12	false	true	0.81	MEDIUM
7	Long Method	62	50	12	false	true	0.81	MEDIUM
8	Long Method	19	13	6	false	true	0.67	LOW
9	Long Method	17	11	6	false	true	0.65	LOW
10	Feature Envy	11	7	4	false	true	0.64	LOW
11	Feature Envy	11	6	5	false	true	0.55	LOW
12	Long Method	44	24	20	false	true	0.55	LOW
13	Long Method	2	1	1	false	true	0.50	LOW

As an example, consider **Index 6**, which corresponds to an instance evaluated for the *Long Method* code smell. A total of 63 votes were cast for this instance, with 51 votes indicating the absence of the smell (negative votes) and 12 votes suggesting its presence (positive votes). The actual classification, or **Ground Truth**, was `false`, indicating that the code smell was not present. However, the model incorrectly predicted `true`, resulting in a false positive. The **Ground Truth Confidence** for this instance was 0.81, corresponding to a `MEDIUM` **Confidence Level**.

We conducted an analysis of the classification errors by categorizing them according to their associated confidence levels. This approach provides a more nuanced understanding of the model's behavior, especially in cases where predictions diverged from the ground truth despite varying levels of certainty. The analysis revealed key insights into the model's strengths and limitations.

High confidence errors (e.g., rows 1–4) involved instances where the ground truth label was associated with a `HIGH` confidence level, yet the model still produced incorrect predictions. These errors underscore the challenges the model faces in accurately distinguishing complex patterns within certain code smells, even when the underlying annotation is highly reliable.

Medium confidence errors (e.g., rows 5–7) suggest that the model's misclassifications may stem from a lack of contextual information or insufficient feature granularity. In these cases, enhancing the input representation or incorporating additional code-level semantics could improve performance.

Low confidence errors (e.g., rows 8–13) reflect substantial uncertainty regarding the correctness of the ground truth itself. These instances likely represent inherently ambiguous or borderline cases, where the model struggles to generalize effectively. They also point to the potential need for more robust training data or improved input encoding strategies to handle such uncertainty.

Together, these observations contribute to a better understanding of where and why the model fails, and offer clear directions for refinement in future iterations.

The analysis of Table 3 underscores the importance of refining the model's input features and voting mechanisms to address misclassifications effectively. Furthermore, improving the handling of low-confidence cases could significantly enhance the overall accuracy of the model for code smell detection tasks.

3.4 Key Results

This subsection synthesizes the primary findings from the evaluation of GPT-4o in identifying Long Method, God Class, and Feature Envy across all research questions (EXP1–EXP4). The results elucidate the model's capabilities and limitations, as well as the implications of integrating code metrics and hyperparameter tuning into the experimental workflow.

Regarding **general trends**, the model exhibited substantial performance improvements from EXP1 to EXP4. Notably, the AUC increased by over 57% (from 0.52 to 0.82), and the F1-Score improved by over 47% (from 0.53 to 0.78) when considering all code smells collectively. The introduction of specialized prompts in EXP2 and the incorporation of code metrics in EXP3 led to a significant reduction in false negatives, thereby enhancing the model's recall—particularly for Feature Envy and Long Method. Although hyperparameter tuning in EXP4 had a relatively modest effect, it slightly improved the overall metrics while preserving the performance gains achieved in EXP3.

For **Long Method**, performance peaked in EXP2, where the model achieved an F1-Score of 0.79 and an AUC of 0.80. These results highlight the effectiveness of using specialized prompts tailored to this smell. However, the integration of metrics in EXP3 introduced a slight decline in performance, with the F1-Score dropping to 0.68 and the AUC to 0.71. This was largely attributed to an increase in false positives caused by the misinterpretation of threshold-based metrics, such as method length. In EXP4, hyperparameter tuning led to modest improvements, raising the F1-Score to 0.70 and the AUC to 0.73, suggesting the potential benefits of further refinement.

In the case of **God Class**, the model consistently achieved perfect results in EXP3 and EXP4, with an F1-Score, AUC, and Accuracy of 1.00 across both experiments. The incorporation of structural metrics—particularly those related to cohesion and dependency—greatly enhanced classification accuracy. These results demonstrate the model's strong ability to identify structural code smells when provided with appropriate contextual and metric-based information.

Feature Envy remained the most challenging smell to classify throughout the study. Despite this, significant progress was observed in EXP3, where the F1-Score increased from 0.45 in EXP2 to 0.65, and Accuracy rose from 0.46 to 0.71. However, the AUC

for Feature Envy peaked at 0.59 in EXP2 and declined to 0.54 in both EXP3 and EXP4, indicating ongoing challenges in achieving a balance between precision and recall. The inclusion of additional metrics, while helpful in some respects, also introduced new false positives, primarily due to the misclassification of legitimate inter-class interactions as instances of Feature Envy.

Overall, GPT-4o detects structural smells (God Class) almost perfectly and simpler smells (Long Method) well, but still struggles with Feature Envy, motivating:

- Enhanced metrics tailored specifically to the characteristics of Feature Envy.
- Additional contextual information or specialized prompts to mitigate false positives and improve classification balance.
- Incorporating iterative optimization techniques to refine the influence of metrics and hyperparameters on prediction performance.

This study provides a strong foundation for leveraging LLMs in automated software quality assessment and offers key insights for future research in improving model robustness and accuracy across diverse codebases.

4 Discussion

This section delves into GPT-4o's performance, highlights potential avenues for enhancement, and evaluates the broader implications for automated code smell detection. The focus is placed on leveraging LLMs for collaborative classification, tailoring specialized prompts, and optimizing metric selection through heuristic methods.

4.1 Effectiveness Across Code Smells

The evaluation demonstrated GPT-4o's ability to effectively identify structural code smells like God Class and simpler smells like Long Method. The integration of code metrics (EXP3) and hyperparameter tuning (EXP4) consistently improved model performance, with God Class achieving perfect results (F1-Score, AUC, and Accuracy of 1.00). These outcomes emphasize the utility of structural metrics such as cohesion and dependency analysis for accurately classifying God Class.

Long Method exhibited stable performance, with hyperparameter tuning (EXP4) slightly enhancing F1-Score and AUC metrics. This suggests that simpler smells can be addressed effectively with concise prompts and basic metrics, although further optimization may yield diminishing returns.

Feature Envy, however, posed notable challenges. Despite significant gains in F1-Score and Accuracy when code metrics were introduced, the AUC remained comparatively low (0.54 in EXP3 and EXP4). These findings highlight the difficulty of capturing nuanced inter-class relationships and emphasize the need for advanced contextual understanding and tailored representations.

4.2 Leveraging LLM Agents for Collaborative Classification

An innovative approach to improving classification accuracy involves employing multiple LLM agents in a collaborative framework. Inspired by human decision-making during code reviews, each agent could specialize in distinct aspects of code smell detection, such as structural properties, inter-class relationships, or historical patterns. These agents could "discuss" and synthesize their perspectives to reach a consensus.

Future research should explore frameworks for agent collaboration, employing ensemble methods, reinforcement learning, or agent-based modeling to optimize decision-making. Collaborative LLM systems could significantly enhance the identification of complex smells like Feature Envy, while integrating developer feedback loops could further improve contextual relevance.

4.3 Specialized Prompts

The findings from EXP3 and EXP4 underscore the importance of specialized prompts tailored to each code smell. For instance, God Class benefited significantly from structural metrics, while Feature Envy required additional contextual information. This demonstrates the inadequacy of universal prompts for addressing the diverse characteristics of different code smells. A modular approach to prompt engineering is therefore recommended. For Long Method, prompts should focus on method length and cohesion-related metrics. In the case of God Class, it is important to incorporate metrics associated with class complexity, cohesion, and dependency analysis. For Feature Envy, prompts should emphasize metrics that capture method-to-class interactions and external dependencies. This tailored strategy has the potential to reduce false positives and improve overall identification performance.

4.4 Optimizing Metric Selection with Heuristic Approaches

Given the complexity and interdependencies among more than 60 metrics, manually identifying the optimal combination for each code smell is impractical. Heuristic methods provide a promising alternative by automating the selection process and systematically exploring the metric space. One such method is the use of **genetic algorithms**, which simulate evolutionary processes to optimize metric selection. These algorithms rely on fitness functions based on performance metrics such as F1-Score or AUC, thereby enabling the identification of metric combinations that maximize classification accuracy for each specific smell.

Other heuristic approaches also hold significant potential. **Simulated annealing** explores the metric space by probabilistically accepting suboptimal solutions during the search process, which helps in avoiding local optima. **Particle swarm optimization (PSO)**, inspired by the collective behavior of swarms, refines metric combinations based on the performance of individual particles in the population. Additionally, **Bayesian optimization** treats performance as a probabilistic function and efficiently navigates the search space to identify optimal metric sets.

Integrating these heuristic techniques into the experimental workflow could lead to more informed metric selection, generate actionable insights, and significantly enhance the overall performance of code smell classification.

4.5 Future Directions

Building on the findings of this study, several future research directions are proposed. One promising avenue involves the development of **collaborative LLM frameworks**, in which multiple large language model agents work together to classify code smells through consensus-based decision-making processes. Another important direction is the advancement of **specialized prompt engineering**, where prompts are tailored to each code smell and incorporate the most relevant metrics to enhance classification accuracy.

In addition, **heuristic metric optimization** should be further explored using techniques such as genetic algorithms, simulated annealing, or Bayesian optimization, allowing for a systematic identification of the most impactful metrics. The adoption of **advanced code representations**, including graph-based or embedding-based models, may also prove beneficial, particularly in capturing the complex dependencies relevant to smells like Feature Envy.

Finally, the **fine-tuning and domain adaptation** of models such as GPT-4o on software engineering-specific datasets could significantly improve their generalization capabilities and robustness, thereby making them more effective for real-world applications in code smell detection.

4.6 Broader Implications

This study highlights the potential of LLMs as adaptive tools for software quality assessment. By incorporating specialized prompts and heuristic optimization, LLMs can complement traditional static analysis tools, offering enhanced capabilities for diverse code smells. The collaborative LLM framework presents an opportunity to revolutionize automated code analysis by mimicking human decision-making processes.

However, challenges remain, including scaling metric selection, managing computational costs, and integrating LLMs into existing workflows. Addressing these limitations through continued research and innovation will be essential to fully realize the transformative potential of LLMs in software engineering.

5 Threats to the Validity

In this section potential threats are outlined and discussed based on the guidelines of Wohlin et al. [25], focusing on study validity and mitigation measures.

5.1 Internal Validity

Internal validity concerns factors that may have influenced the results without being accounted for. One such factor is the **metric selection bias**, as the study relied on over 60 quality metrics. Including all metrics without careful optimization may have introduced noise, especially for smells such as Long Method, where many metrics had limited impact. Future work could address this by employing heuristic methods, such as genetic algorithms, to systematically identify the most impactful metrics. Another factor is the **influence of prompt engineering**. The performance differences observed across code smells suggest that the quality of prompt design significantly affects outcomes. This variability might obscure the inherent capabilities of GPT-4o, and thus, future studies should aim to standardize prompts while exploring optimal configurations for each specific smell. Lastly, **dataset composition** poses a concern, as the study focused on a limited number of projects, predominantly from the "jasml-0.10" system. This narrow scope may have biased the results by limiting the diversity of code patterns and smells encountered by the model.

5.2 External Validity

External validity addresses the generalizability of the findings to broader contexts. One limitation in this regard is the presence of **language and domain constraints**, as the study primarily utilized datasets written in Java. This focus may restrict the applicability of the findings to other programming languages or paradigms. To enhance generalizability, future studies should consider including languages with different syntactic and structural characteristics, such as Python or C++. Another concern is the **generalizability of the dataset** itself. Although the quality metrics and code smells analyzed are commonly found in software engineering, the limited scope of datasets used may not capture the full variety of real-world software systems. Therefore, incorporating larger and more diverse datasets in future work is essential for validating and strengthening the conclusions.

5.3 Construct Validity

Construct validity evaluates whether the study accurately measured what it intended to measure. One relevant factor is the **definition of code smells**. The study adopted specific definitions for Long Method, God Class, and Feature Envy, which, although consistent with the literature, may not fully capture the nuances of these smells in every context. Future research could explore alternative or expanded definitions to improve robustness. Another consideration is the choice of **evaluation metrics**. The study primarily relied on F1-Score, AUC, and Accuracy, which, while widely used, may not encompass all dimensions of model performance. Additional measures such as Precision-Recall Curves or domain-specific metrics could offer deeper insights into model behavior. Finally, the study faces limitations related to **LLM contextual constraints**. Although GPT-4o effectively leveraged prompts and static code metrics, the absence of dynamic context—such as runtime behavior—may hinder its capacity to accurately detect smells like Feature Envy. This challenge could be addressed in future work through the incorporation of dynamic analysis techniques or graph-based code representations.

5.4 Conclusion Validity

Conclusion validity pertains to the reliability of the conclusions drawn from the study. One important consideration is the **result variability**, as the observed differences in performance across code smells suggest that the conclusions might be sensitive to specific configurations or datasets. Conducting repeated experiments with alternative datasets and varied prompt designs would help reinforce the robustness of the findings. Another factor is the issue of **algorithmic generalization**. Since the results are based solely on GPT-4o, they may not necessarily extend to other large language models or traditional machine learning approaches. To better contextualize the results, future studies should include comparisons

with different LLMs and conventional tools. Lastly, the lack of implemented **heuristic integration**—despite being proposed as a strategy for metric optimization—means that the current conclusions regarding metric relevance might not fully represent optimal configurations. Incorporating such heuristics in follow-up research would provide a more complete assessment.

5.5 Mitigation Strategies

Several measures have been proposed or implemented to address these threats. These include expanding the datasets to encompass more diverse projects and programming languages, adopting heuristic methods to identify the most relevant metrics for each code smell, and exploring advanced prompt engineering techniques while standardizing configurations across experiments. Additionally, benchmarking GPT-4o against other models and tools can help validate its performance in broader contexts. The incorporation of additional evaluation metrics is also essential to capture more nuanced aspects of identification performance. Despite these limitations, the study offers valuable insights into the potential and challenges of using large language models for code smell identification. Addressing the outlined threats will be crucial for future work to build upon these findings and further refine the proposed approach.

6 Related Work

Code Smell Identification with Deep Learning. Zhang et al. [29] present an approach named *DeleSmell*, which leverages deep learning models and latent semantic analysis (LSA) for code smell identification. This methodology incorporates convolutional neural networks (CNN), gated recurrent units (GRU), and support vector machines (SVM) to improve identification precision and accuracy. The experimental results indicate an improvement in the identification accuracy up to 4.41% compared to the existing methods.

Lin et al. [12] were pioneers in applying convolutional neural networks to code smell identification based on semantic features, building on prior research by [13]. Their solution demonstrated satisfactory performance in addressing uncontrolled side effects and contrived complexity, achieving an F1-Score exceeding 0.75.

Yu et al. [27] introduced a graph neural network (GNN)-based approach specifically to identify the feature envy code smell, using code metrics and method-calling relationships represented as a graph. In experiments across five open-source projects, that approach achieved an average F1-Score of 78.90%, surpassing comparative methods by 37.98%.

Wu et al. [26] proposed an ensemble method that uses a Mixture of Experts (MoE) architecture, integrating various code smell identification toolsets to enhance identification and refactoring capabilities. The MoE model is trained to select the most suitable expert tool for different code smells and amalgamates their results, thereby improving the refactoring performance of LLMs. The evaluation focuses on complex code smells, including Refused Bequest, God Class, and Feature Envy.

Code Smell Identification with LLMs. Lucas et al. [14] investigate the effectiveness of LLM to automatically identify test smells, which are coding issues that arise from inadequate practices during software development. The research evaluated ChatGPT-4, Mistral, and Gemini on 30 types of test smells across various codebases in seven programming languages. The results showed that ChatGPT-4 identified 21 types of test smells, Gemini identified 17 types, and Mistral found 15 types.

Silva et al. [20] present an initial exploration of ChatGPT's effectiveness in identifying code smells in Java projects, utilizing a large dataset with four specific smells (Blob, Data Class, Feature Envy, and Long Method) classified by severity. Two prompts were tested: a generic one to assess overall identification capabilities and a specific one tailored to the classified smells. As a result, the evaluation metrics precision, recall, and F-measure, indicated that ChatGPT's accuracy improves significantly with the specific prompt, being 2.54 times more likely to yield correct results.

7 Conclusion

This study evaluated the capability of a state-of-the-art LLM, the GPT-4o model, to identify three code smells (Long Method, God Class, and Feature Envy) using specialized prompts and incorporating code metrics into the classification process. By leveraging a refined code smell oracle adapted from Reis et al. [17], the model's performance was systematically assessed across different research questions, culminating in a comprehensive evaluation of its strengths and limitations.

Our analysis revealed significant improvements across all metrics when progressing from EXP1 to EXP4, with overall performance increasing by 64% in AUC (from 0.50 to 0.82) and 56% in F1-Score (from 0.50 to 0.78). God Class consistently achieved the highest identification performance, attaining perfect scores (Accuracy, F1-Score, and AUC) in EXP4. Long Method demonstrated steady improvements, achieving an F1-Score of 0.76 and an AUC of 0.78. Feature Envy, while showing gains, remained the most challenging, with an F1-Score of 0.65 and an AUC of 0.54. Interestingly, the study found that hyperparameter tuning had negligible impact on performance across all code smells. The improvements observed were primarily driven by the inclusion of relevant code metrics, reaffirming the importance of tailored prompts and metric selection over parameter optimization. Future directions include heuristic optimization for metric selection, collaborative LLM frameworks, and the development of advanced representations to fully realize the potential of LLMs in software quality assessment.

ARTIFACT AVAILABILITY

All artifacts, including datasets, scripts, and documentation from this study are available in our replication package [8].

ACKNOWLEDGMENTS

This work received partial funding from CNPq-Brazil, Universal grant 404406/2023-8, and support from CAPES - Funding Code 001.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Lucas Aguiar, Matheus Paixao, Rafael Carmo, Matheus Freitas, Eliakim Gama, Antonio Leal, and Edson Soares. 2024. Multi-language Software Development in the LLM Era: Insights from Practitioners' Conversations with ChatGPT. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 489–495.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, and Baishakhi Ray. 2022. Multitask Learning for Code Review and Bug Detection Using Large Language Models. *arXiv preprint arXiv:2204.01875* (2022). https://arxiv.org/abs/2204.01875

[4] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[5] Checkstyle. 2024. *Checkstyle Documentation*. https://checkstyle.sourceforge.io/

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. In *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2107.03374

[7] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[8] Anderson Gomes, Denis Sousa, Paulo Maia, and Matheus Paixao. 2025. Attentionsmelling: Using Large Language Models to Identify Code Smells. https://github.com/denisousa/Attentionsmelling-Using-Large-Language-Models-to-Identify-Code-Smells

[9] Taojun Hu and Xiao-Hua Zhou. 2024. Unveiling LLM Evaluation Focused on Metrics: Challenges and Solutions. *arXiv preprint arXiv:2404.09135* (2024).

[10] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2024. Improved Program Repair Methods using Refactoring with GPT Models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 569–575.

[11] Haoran Li et al. 2022. Coder: A Contextual Language Model for Code Documentation. *arXiv preprint arXiv:2203.11329* (2022). https://arxiv.org/abs/2203.11329

[12] Tao Lin, Xue Fu, Fu Chen, and Luqun Li. 2021. A novel approach for code smells detection based on deep leaning. In *Applied Cryptography in Computer and Communications: First EAI International Conference, AC3 2021, Virtual Event, May 15-16, 2021, Proceedings 1*. Springer, 171–174.

[13] Tao Lin, Jianhua Gao, Xue Fu, and Yan Lin. 2015. A novel bug report extraction approach. In *Algorithms and Architectures for Parallel Processing: ICA3PP International Workshops and Symposiums, Zhangjiajie, China, November 18-20, 2015, Proceedings 15*. Springer, 771–780.

[14] Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. 2024. Evaluating Large Language Models in Detecting Test Smells. *arXiv preprint arXiv:2407.19261* (2024).

[15] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*. Springer, 387–402.

[16] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[17] José Pereira dos Reis, Fernando Brito e Abreu, and Glauco de Figueiredo Carneiro. 2022. Crowdsmelling: A preliminary study on using collective knowledge in code smells detection. *Empirical Software Engineering* 27, 3 (2022), 69.

[18] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 151–160.

[19] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 313–322.

[20] Luciana Lourdes Silva, Jânio Silva, João 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*. 400–406.

[21] SonarSource. 2024. *SonarQube Documentation*. https://www.sonarqube.org

[22] Christophe Tribes, Sacha Benarroch-Lelong, Peng Lu, and Ivan Kobyzev. 2023. Hyperparameter optimization for large language model instruction-tuning. *arXiv preprint arXiv:2312.00949* (2023).

[23] Siyin Wang, Shimin Li, Tianxiang Sun, Jinlan Fu, Qinyuan Cheng, Jiasheng Ye, Junjie Ye, Xipeng Qiu, and Xuanjing Huang. 2024. LLM can Achieve Self-Regulation via Hyperparameter Aware Generation. *arXiv preprint arXiv:2402.11251* (2024).

[24] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[25] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[26] 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*. 1345–1357.

[27] Dongjin Yu, Yihang Xu, Lehui Weng, Jie Chen, Xin Chen, and Quanxin Yang. 2022. Detecting and refactoring feature envy based on graph neural network. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 458–469.

[28] Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. 2023. Automl-gpt: Automatic machine learning with gpt. *arXiv preprint arXiv:2305.02499* (2023).

[29] Yang Zhang, Chuyan Ge, Shuai Hong, Ruili Tian, Chunhao Dong, and Jingjing Liu. 2022. DeleSmell: Code smell detection based on deep learning and latent semantic analysis. *Knowledge-Based Systems* 255 (2022), 109737.

[30] Mengyuan Zhu, Jiawei Wang, Xiao Yang, Yu Zhang, Linyu Zhang, Hongqiang Ren, Bing Wu, and Lin Ye. 2022. A review of the application of machine learning in water quality evaluation. *Eco-Environment & Health* 1, 2 (2022), 107–116.