Beyond Strict Rules: Assessing the Effectiveness of Large Language Models for Code Smell Detection

Saymon Souza, Amanda Santana, Eduardo Figueiredo,
Igor Muzetti, João Eduardo Montandon, and Lionel Briand

***Abstract*—Code smells are symptoms of potential code quality problems that may affect software maintainability, thus increasing development costs and impacting software reliability. Large language models (LLMs) have shown remarkable capabilities for supporting various software engineering activities, but their use for detecting code smells remains underexplored. However, unlike the rigid rules of static analysis tools, LLMs can support flexible and adaptable detection strategies tailored to the unique properties of code smells. This paper evaluates the effectiveness of four LLMs – DeepSeek-R1, GPT-5 mini, Llama-3.3, and Qwen2.5-Code – for detecting nine code smells across 30 Java projects. For the empirical evaluation, we created a ground-truth dataset by asking 76 developers to manually inspect 268 code-smell candidates. Our results indicate that LLMs perform strongly for structurally straightforward smells, such as *Large Class* and *Long Method*. However, we also observed that different LLMs and tools fare better for distinct code smells. We then propose and evaluate a detection strategy that combines LLMs and static analysis tools. The proposed strategy outperforms LLMs and tools in five out of nine code smells in terms of F1-Score. However, it also generates more false positives for complex smells. Therefore, we conclude that the optimal strategy depends on whether Recall or Precision is the main priority for code smell detection.**

***Index Terms*—Code Smells, LLM, Empirical Study.**

I. Introduction

Code smells are symptoms or indicators of potential code quality problems that may affect software maintainability and reliability [1]. Code maintainability is essential because it refers to how easily code can be understood, changed, and improved [2], [3]. In this context, previous research has shown that code smells can increase development costs [4], [5], reduce software reliability [6], [7], and lead to software defects [8]–[11]. Detecting and refactoring code smells can be challenging because manual reviews require advanced skills, are expensive and slow, while automated tools are imprecise and still require human interpretation [5]. Common examples of code smells include methods that become complex because they take on too many responsibilities, intense communication between classes, or the same code snippets repeated in different places [1].

Recent advances in Large Language Models (LLMs) have sparked interest in their use for coding problems [12]–[14]. In fact, these models have shown promising capabilities for generating code [15], repairing bugs [16], and supporting software testing [17], but the challenges of using LLMs to detect code smells remain underexplored and lack relevant benchmarks [18]. In contrast to traditional automated detection tools that adhere to strict rules, such as JDeodorant [19] and PMD [20], LLMs can provide an innovative approach to detecting code smells. Their ability to understand complex contexts may enable them to adopt flexible, adaptable strategies for different types of code smells. In fact, some preliminary studies have investigated the use of LLMs to detect and refactor code smells [18], [21], [22]. However, to the best of our knowledge, they do not provide strong empirical evidence to support, for instance, the claim that LLMs perform better than traditional code smell detection tools.

Moreover, early investigations of LLMs for code smell detection typically employ a small set of smell types and simple code samples [23]–[25]. Thus, it is essential to further investigate the effectiveness and limitations of LLMs for detecting code smells in real-world software projects [18]. Such software projects introduce numerous challenges for LLMs, as they require navigating large, complex codebases, adhering to diverse coding standards, and ensuring compatibility with existing systems [26]. These characteristics make automated detection of code smells particularly difficult, especially when compared to controlled or synthetic code examples. Evaluating how effectively LLMs can detect code smells under these realistic conditions is therefore crucial, as it helps determine whether they can serve as practical alternatives, or valuable complements, to traditional detection tools [21]. More importantly, we need to empirically determine not only how LLMs compare with other automated techniques, but also how closely their detections align with human judgments of what constitutes a code smell.

This paper evaluates the effectiveness of LLMs by using four models, namely DeepSeek-R1, GPT-5 mini, Llama-3.3, and Qwen2.5-Coder, to detect nine code smells in 30 real-world Java projects. Our goal is to understand which code smells LLMs successfully detect and when they fail. To perform our empirical evaluation, we first expanded a large dataset of 30 top-starred Java projects mined from GitHub [27] with LLM and human evaluations. We then created ground truth by having 76 developers manually inspect 268 code-smell candidates.

Using the newly expanded dataset as the foundation of our investigation, we conducted three complementary analyses to assess the effectiveness of LLMs in detecting code smells. The main analysis evaluates how accurately each LLM identifies code smells relative to the human-validated ground truth derived from developer evaluations. Building on this foundation, the second analysis compares LLM performance with that of traditional static analysis tools to identify where LLMs offer advantages or show limitations. Finally, the third analysis

examines whether combining both detector types using a voting-based strategy can further improve detection accuracy and coverage.

Our results indicate that LLMs perform strongly for structurally simple smells, such as *Large Class* and *Long Method*, especially when multiple detection strategies are combined via voting. This proposed strategy that aggregates outputs of four LLMs and two static analysis tools maximizes recall and F1-score. For more subjective or context-dependent smells, such as *Feature Envy* and *Refused Bequest*, LLMs provide mixed results, although specialized tools or carefully selected individual LLMs remain the most effective. The results also show that no single strategy outperforms the others across all smells. In most cases, the optimal strategy varies depending on the smell type. In particular, the combined strategy consistently improves recall and provides robust detection of common, easily quantifiable smells, making it an attractive choice for practitioners seeking greater code-smell coverage. However, this combined strategy may generate more false positives for complex smells. As a result, the choice between a combined and an individual detection strategy should reflect the project's tolerance for such trade-offs and its specific code quality goals.

The contributions of this work can be summarized as follows.

- We created a ground truth of code smells in 30 real-world Java projects mined from GitHub. This ground truth can be used to compare the effectiveness of different strategies to detect code smells, including other static analysis tools and LLMs beyond those evaluated in our study.
- We demonstrate that a combined strategy offers the best recall and F1-score for detecting 5 out of 9 code smells. However, specialized tools or LLMs remain preferable for more subjective cases, underscoring the importance of selecting detection strategies based on the smell type and the desired trade-off between Recall and Precision.
- For software developers, we indicate which automated strategy is more effective in detecting each type of code smell. Our results also help teams refine their workflows by balancing Recall, Precision, and review costs.
- We provide a dataset and all scripts to replicate and expand this study, for instance, with other artificial intelligence models and tools. The online artifacts used in this study are available in our replication package [28].

The remainder of this paper is organized as follows. Section II introduces code smells and describes the previous dataset, which is extended in our study. Section III outlines how we extended the previous dataset and created our ground truth. Section IV explains our research method, and Section V presents our empirical results. Section VI analyzes the main findings of this study. Section VII discusses possible threats to validity and the actions we took to mitigate them. Section VIII discusses some related work. Finally, Section IX provides our final thoughts and suggests ideas for future research work.

II. Background

This section presents an overview of the code smells analyzed and the dataset used in this study.

A. Code Smells and Detection Techniques

Code smells are symptoms or indicators of potential code quality degradation that may affect the software maintainability and reliability [1]. They not only impact code understandability, reusability, and extensibility [2], [6], but they may also be the source of bugs and code instability [8], [11], [27], [29]. Given their potential to increase development costs [2], it is important to identify code smells and refactor them. *Refactoring* is an activity in which the code is modified to improve its internal quality without changing its external behavior [1]. Table I describes the nine code smells used in this study. The first column shows the smell name, while the second column briefly defines each code smell. More details on their definition can be found in the books of Fowler [1], and Lanza and Marinescu [30]. These particular code smells were chosen because they are well supported by detection tools [19], [20] and cover a broad range of modularity-related issues.

TABLE I
Code Smell Definitions

Code Smells	Definitions
Data Class	A class composed mainly of fields and getter/setter methods, with little or no meaningful behavior. [1]
Dispersed Coupling	A method that depends on many other classes, but with low coupling intensity. [30]
Feature Envy	A method that accesses members of other classes more than its own. [1]
Intensive Coupling	A method that heavily interacts with one or a few other classes, forming a tight cluster. [30]
Long Method	A method that is excessively long or takes on too many responsibilities. [1]
Large Class	A class that handles multiple responsibilities or contains many lines of code. [1]
Long Parameter List	A method signature that requires an excessive number of parameters. [1]
Refused Bequest	A subclass that overrides or ignores most inherited behavior, indicating poor inheritance fit. [1]
Shotgun Surgery	A change in one module forces many small changes scattered across other modules. [1]

Several techniques to detect code smells have been proposed in the literature, such as manual code inspection [31], [32], static analysis tools [19], [20], refactoring opportunities [33], change history analysis [34], and machine learning models [29], [32], [35], [36]. Although developers widely use static analysis tools [19] to detect code smells, several studies in the literature indicate they have poor agreement with developers' perception of what a code smell is [37], [38]. Previous work [26], [35], [36] also indicates the low effectiveness of some classic machine learning models, such as Naive Bayes, Decision Tree, and Random Forest, to detect code smells. More importantly, although some recent preliminary studies have been published on these code smell detection techniques [18], [22], we still lack strong empirical evidence on the effectiveness of LLMs in supporting code smell detection.

B. Datasets of Code Smells

Several datasets of code smells are available in the literature [32], [35], [39]. However, they have several limitations. For instance, systems in some datasets may not reflect current

development practices [35], [39], and they exhibit limited coverage of code smells [32]. To avoid these limitations, we have used and extended our previous dataset [27]. Our dataset extension includes additional features, the integration of LLMs for code smell detection, and human evaluations (see Section III). The selected dataset included 3,459 instances of nine code smells (three at the class level and six at the method level) across 30 open-source Java systems on GitHub. We focus on Java projects for this study because many tools are available to detect a wide range of code smells in this programming language [37].

Table II provides details about the 30 systems included in the used dataset [27]. The first column lists the names and versions of each system. The following three columns show the number of classes (NOC), the number of methods (NOM), and the lines of code (LOC) for each system, respectively. The last column (Stars) displays the total number of stars for each repository. The following criteria were used to select the systems: (i) they were among the top-stared Java systems on GitHub; (ii) they had commits merged in the last two years; and (iii) they cover different domains, sizes, and levels of maturity. Systems for educational purposes were excluded [27]. This variety of systems helps us investigate how well LLMs detect code smells across software systems of different domains and sizes.

TABLE II
DATASET DESCRIPTION

Name	NOC	NOM	LOC	Stars
arthas-3.4.3	834	4,733	39,973	36,580
cryptomator-1.6.1	590	2,690	16,350	13,388
dbeaver-21.0.2	6,449	36,575	348,608	44,710
easyexcel-2.2.11	249	1,629	10,639	33,549
elasticsearch-analysis-ik	28	203	2,051	17,200
fastjson-1.2.76	249	1,996	44,434	25,756
gson-2.8.8	231	924	11,721	23,925
guava-30.1.1	27,412	200,616	2,125,859	50,993
HikariCP-4.0.0	68	581	4,530	20,634
hutool-5.7.17	1,214	11,966	79,432	29,969
java-faker-1.0.2	105	751	3,602	4,889
jedis	749	6,219	27,404	12,132
jenkins-2.287	2,432	14,775	120,382	24,269
jitwatch-1.4.2	538	7,346	46,527	3,185
jsoup-1.14.2	246	1,551	17,449	11,235
junit4-4.13.2	310	1,541	10,769	8,533
libgdx-gdx-1.9.14	2,714	39,338	208,028	1,263
mall-1.0.2	747	14,322	100,990	81,223
mybatis-3.5.6	378	2,582	20,533	20,179
nanohttpd-2.3.1	75	405	3,821	7,129
netty-socketio-1.7.18	138	712	5,217	6,980
redisson-3.15.3	1,613	13,607	82,104	23,946
retrofit-1.6.0	118	403	4,790	43,653
rocketmq-4.9.2	996	7,536	70,871	21,990
Sa-Token-1.28.0	191	1,600	8,848	18,043
Sentinel-1.8.3	1,029	4,963	41,366	22,852
spring-cloud-alibaba-2.2.2	411	2,003	13,594	28,662
webmagic-develop-0.7.6	207	955	6,757	11,600
xxl-job-2.3.0	150	741	8,374	29,124
zxing-3.4.1	303	1,783	23,614	33,495
Total	50,774	385,046	3,508,637	682,312

Table III lists the four static analysis tools used to detect the nine different types of code smells [27]: *Data Class* (DC), *Dispersed Coupling* (DiCo), *Feature Envy* (FE), *Intensive Coupling* (IC), *Long Method* (LM), *Large Class* (LC), *Long Parameter List* (LPL), *Refused Bequest* (RB) and *Shotgun Surgery* (SS). A "✔" mark in the table represents the tool of choice for detecting a specific smell. For instance, we used JDeodorant and JSpIRIT to detect *Large Class*. Individual outputs of the four static analysis tools used in our previous study are also available [27]. We use them in this paper to support comparisons between LLMs and static analysis tools. We chose these tools because they have shown good accuracy in detecting code smells in previous studies [29], [32], [33], [35], [37], [40].

TABLE III
STATIC ANALYSIS TOOLS USED TO DETECT CODE SMELLS

Tool	DC	DiCo	FE	IC	LC	LM	LPL	RB	SS
JDeodorant			✔		✔	✔			
PMD	✔						✔		
Organic	✔	✔	✔	✔		✔	✔	✔	✔
JSpIRIT		✔		✔	✔			✔	✔

III. AN EXTENDED DATASET OF CODE SMELLS

Figure 1 presents the steps we followed in this paper. Steps 1 to 5 outline how we extended the previous dataset [27] by incorporating LLM-generated code-smell information for this study. The following sections describe these steps.

A. Selected Large Language Models

The first step of our study (Step 1 in Figure 1) is to select the LLMs. Table IV shows important information about the LLMs selected in this study: OpenAI's GPT-5 mini, Meta's Llama-3.3-70B-Instruct, DeepSeek's DeepSeek-R1-Distill-Qwen-32B, and Qwen's Qwen2.5-Coder-32B-Instruct. All models have similar features regarding context windows and knowledge cutoffs. For instance, DeepSeek-R1 and Qwen2.5-Coder both feature 32.5 billion parameters and a large context window of 131,000 tokens. Their knowledge cutoffs are July 2024 and March 2024, respectively. Llama-3.3 has 70 billion parameters and a context window of 128,000 tokens. Although GPT-5 mini does not reveal its parameter count, it has a similar context window of 128,000 tokens.

TABLE IV
SELECTED LARGE LANGUAGE MODELS

Model	Parameters	Context Window	Knowledge Cutoff
DeepSeek-R1	32.5B	131K	July 2024
GPT-5 mini	N/A	128K	May 2024
Llama-3.3	70B	128K	Dec 2023
Qwen2.5-Coder	32.5B	131K	Mar 2024

We chose these LLMs because they are widely used in recent research [41]–[45]. For instance, GPT models have attracted a lot of attention from researchers, with studies covering many topics of software engineering, such as code generation [25], computer science education [45], refactoring [42],

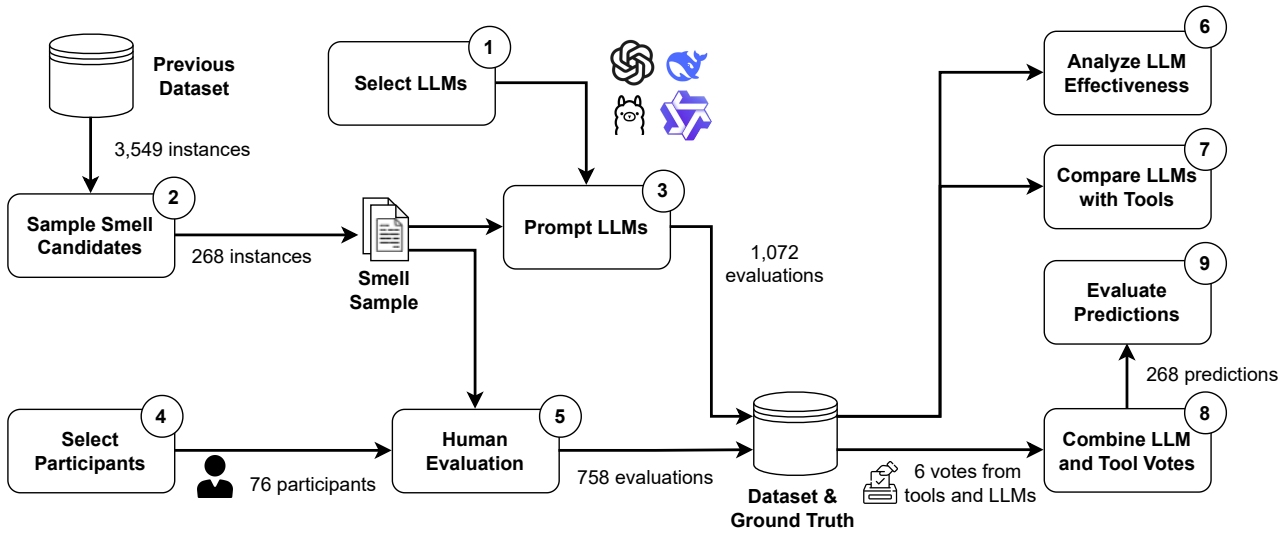

Fig. 1. Steps of our study.

code review [46], test case generation [47], and library selection [44]. We also included Llama-3.3 and DeepSeek-R1 because they are open source, allowing us to compare them with closed-source models, such as GPT-5 mini. For instance, in December of 2025, developers downloaded DeepSeek-R1 over 1.6 million times on HuggingFace, while Llama-3.3 reached more than 600,000 downloads, demonstrating strong interest in these models from the developer community. In addition, we included the code-generation model Qwen2.5-Coder to provide a different perspective on our study. Qwen2.5-Coder was the highest-ranked code model on Hugging Face's leaderboard in December of 2025 [1].

B. Sampling Code Smells

The second step of our study (Step 2 in Figure 1) was sampling code smells. We randomly selected 268 instances of code smells as candidates for this study. We defined the sample size based on three factors: balance, representativeness, and the viability of human validation. For representativeness and balance, we selected the same number of smells for each system and smell type in our empirical study. That is, we selected one instance of each smell in each system; 9 * 30 = 270, but we removed two duplicates. The duplicates have the same code in the methods of different systems. The sample size was determined by constraints on enrolling a large group of participants to manually evaluate the code smell instances.

The selected sample includes both smelly and non-smelly classes and methods [27]. These groups are balanced to allow both static analysis tools and LLMs to distinguish smelly from non-smelly instances. Although six types of code smells in this study are found at the method level, we decided to provide the entire class as a context, including the smelly methods, to the LLMs. This decision gives the models more context to help them detect code smells. Such context information could be particularly important for some smells that rely on inheritance relationships (e.g., *Refused Bequest*) and class dependencies (e.g., *Shotgun Surgery*).

[1]https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard

C. The Used Prompt

Step 3 in Figure 1 is prompting the four LLMs to detect code smells. Figure 2 shows the structured prompt template that we used in this step, and Figure 3 shows an example of our prompt for *Large Class*. We rely on Chain-of-Thought (CoT) prompting [48], which involves a series of intermediate reasoning steps. This prompting strategy aims to improve a model's capacity to produce organized, insightful answers. To guide the models' reasoning, we formulated four questions for each code smell in our study, using the definitions and detection strategies provided by Lanza and Marinescu [30]. Except for *Long Parameter List*, each smell corresponds to a dedicated section in the Lanza and Marinescu book [30], which we use to generate the guiding questions. For *Long Parameter List*, we adopted the same approach but relied on a research paper that specifically focuses on its detection strategies [49]. In summary, this step ensures that the detection strategies in our prompts align with the established literature-based criteria for each code smell in our study.

The gray boxes in Figure 3 present the quotes from the Lanza and Marinescu book [30] for the Large Class code smell, while the box on the left shows the derived questions we used in our prompt. More precisely, Section 5.3 of their book [30] proposes a detection strategy for this smell by identifying three key symptoms: (1) "the class centralizes system intelligence, or its functional complexity is very high", (2) "the class cohesion is low", and (3) "the class uses many attributes from other classes". Each symptom is accompanied by detailed explanations summarized in the first three items of Figure 3. In addition, Lanza and Marinescu [30] provide a general description of the smell, which contributes to a more comprehensive understanding. This general description motivated the inclusion of an additional, broader question,

Prompt

You are a software expert analyzing one Java file for symptoms that may indicate the `[Smell name]` code smell.

`[Description of the smell]`

Please answer the following questions step by step before reaching a conclusion about the code smell.

`[Chain of thoughts with steps to detect each specific smell]`

Instructions:
Based on the step-by-step questions above, you should indicate if the following Java file has the code smell. Please start your answer with "YES, I found `[Smell name]`" if you detect symptoms that could indicate this smell, or "NO, I did not find `[Smell name]`" if you do not. Briefly explain your reasoning, answer the questions and provide the summary.

`[Java source code with the smell]`

Fig. 2. Prompt template used for detecting code smells

which is illustrated by item (4) in Figure 3. The prompts we used for the other eight smells are available in our replication package [28]. We systematically applied this process to both derive detailed detection criteria and define the questions used across all studied code smells.

Because GPT-5 mini does not support custom temperature settings [2], we used its default temperature of 1.0. To ensure consistency and facilitate fair comparison across all models, we applied this same temperature setting to all four models in our study. Other empirical studies [50]–[52] in software engineering used similar temperature values. Finally, we limited all LLMs' outputs to 1,500 characters to simplify our analysis.

D. Ground Truth Creation

The goal of our ground truth is to represent the human perspectives of the analyzed code smells. As shown in Step 4 of Figure 1, we first selected 76 participants to manually evaluate the 268 code smell candidates. These participants are undergraduate Computer Science students in their final year. A background assessment confirmed that they had sufficient experience with Java programming. This assessment asked participants to indicate which Software Engineering topics they were familiar with from a predefined list, describe their professional experience, and self-assess their proficiency in both Java programming and code smell detection. Skill levels were rated on a five-point scale, ranging from (1) "Never heard of it" to (5) "I'm an expert". A complete breakdown of the participants' demographics and the forms used in the empirical experiment is available in our replication package [28]. We then offered participants a preparatory 1-hour lecture to ensure they had a clear understanding of code smells before participating in the evaluation. After finishing data collection, we created nine individual ground truths of manually validated classes, one for each code smell selected for this study. Each ground truth is represented by a distinct subset of 29 or 30 instances, depending on the code smell.

In Step 5, for each code smell, we asked participants to rate a class code on a five-point scale from 1 (it is definitely not a smell) to 5 (it is definitely a smell). Each participant evaluated only 10 randomly sampled classes for different code smells, yet two participants evaluated nine classes each. It is important to note that at least two participants evaluated each class. We then consider a true positive in the ground truth if the average evaluator score exceeds 3. Otherwise, the class is considered non-smelly. To ensure valid comparisons, all LLMs, human annotators, and static analysis tools detect the same set of instances (see Section III-B).

IV. Research Method

This section details the research questions of our study. It also describes the evaluation steps taken to assess LLMs for code smell detection and details the data analysis methods used to interpret our results.

A. Goal and Research Questions

This study aims to evaluate the effectiveness of LLMs in detecting code smells in software projects. We also explored whether a combined detection strategy incorporating LLM and tool outputs can serve as a reliable alternative to code smell detection. To achieve these goals, we defined the following three research questions (RQs).

- **RQ1:** How effective are LLMs in detecting code smells?
- **RQ2:** How do LLMs compare with static analysis tools in code smell detection?
- **RQ3:** How effective is automated code smell detection when combining outputs from LLMs and tools?

B. Evaluation Steps

Steps 6 to 9 in Figure 1 depict the analyses we performed to answer the research questions in this study. In these steps, we rely on the dataset and corresponding ground truth (see Section III-D) to evaluate the effectiveness of LLMs for code smell detection using three metrics: Recall, Precision, and F1-Score [53]. To answer RQ1, Step 6 directly compares the LLM outputs with the ground truth created from 758 human evaluations.

In Step 7 of Figure 1, we answer RQ2 by comparing the effectiveness of LLMs with the static analysis tools used as baselines. The effectiveness of each LLM and tool is measured by its alignment with the human perspective on the analyzed code smells. Finally, we use Recall, Precision, and F1-Score to determine whether a combined strategy presented in Section IV-C is an effective solution for code smell detection.

[2]https://platform.openai.com/docs/guides/latest-model

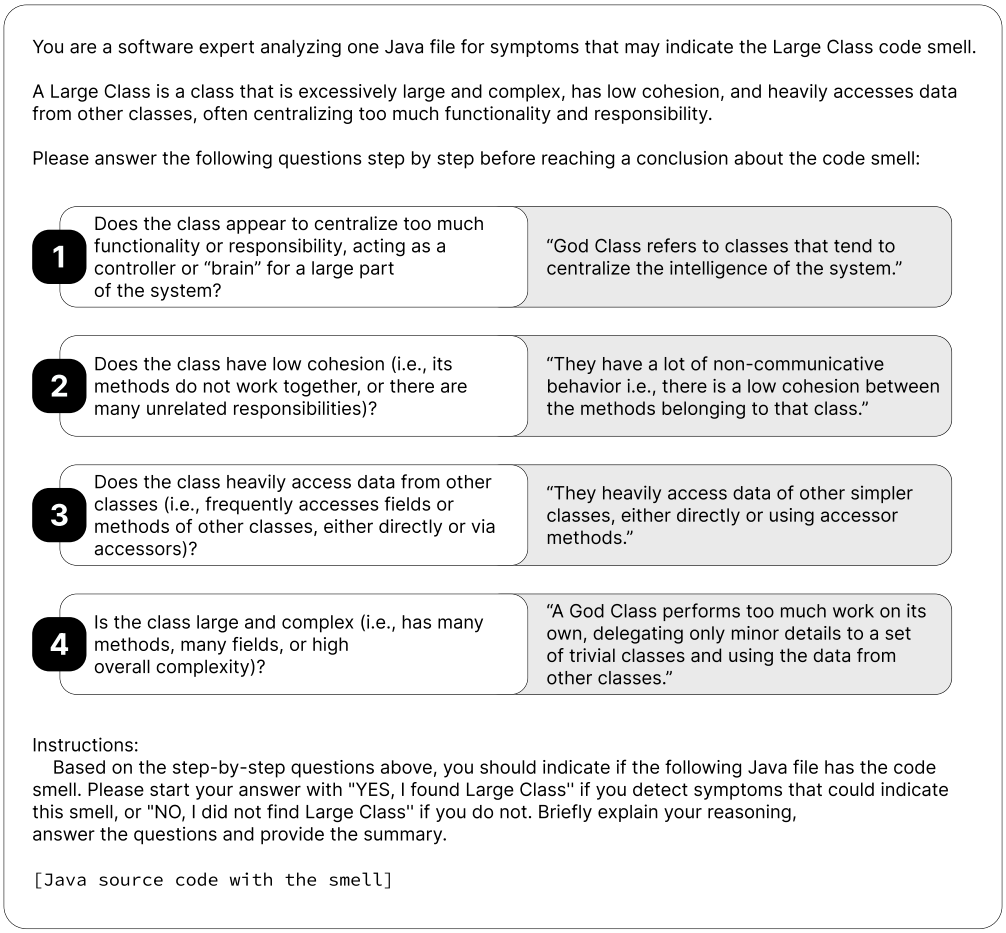

Fig. 3. Prompt used to detect Large Class

C. Combining Outputs of Tools and LLMs

We propose a combined strategy in Step 8 of Figure 1, using a voting mechanism that takes as input the scores from the tools and LLMs used in this study. More precisely, this voting strategy [54] combines the outputs of each code smell from two static analysis tools and four LLMs (6 votes). We should note that, although we rely on the outputs of four static analysis tools, only two of them were used for each smell, as detailed in Section II-B. A code smell was predicted if it received at least three votes for that smell. We chose this number after empirically evaluating the results with different values and observing a higher correlation between the ground truth and the combined predictions. All votes carry equal weight, ensuring that both tools and LLMs have the same influence on the final decision regarding a smell candidate.

Table V summarizes the sample of code smell candidates from the used dataset (2nd column), the identified true positives in the ground truth (3rd column), and the detected ones by the proposed strategy (last column). Our results reveal an imbalance in the number of code smells, consistent with previous studies [33], [37], [40], [55]. Notably, combined predictions generally flag more code smells than human evaluators. For instance, in six out of the nine smells, the number of instances detected in the combined predictions exceeds the manually annotated ground truth. However, easily detectable code smells, such as *Long Method* and *Large Class*, are commonly present in both the ground truth and the combined results. In contrast, more complex smells, such as *Refused Bequest* and *Shotgun Surgery*, are less represented. This distribution highlights the persistent challenges of reliably detecting certain types of code smells, whether evaluated by humans or automated strategies.

TABLE V
CODE SMELLS IN THE DATASETS

Code Smell	Sample	Ground Truth	Combined Strategy
Data Class	29	17	21
Dispersed Coupling	30	18	22
Feature Envy	29	16	24
Intensive Coupling	30	19	25
Large Class	30	20	23
Long Method	30	25	22
Long Parameter List	30	16	25
Refused Bequest	30	10	10
Shotgun Surgery	30	16	16

V. RESULTS

This section presents the main findings of this paper, focusing on the most interesting results. The section is structured according to our three research questions.

TABLE VI
EFFECTIVENESS METRICS FOR THE LLM AND TOOLS USING SOFTWARE DEVELOPER EVALUATIONS

		Large Language Models				Static Analysis Tools			
Smell	**Metrics**	**DeepSeek-R1**	**GPT-5 mini**	**Llama-3.3**	**Qwen2.5-Coder**	**JDeodorant**	**JSpIRIT**	**Organic**	**PMD**
Data Class	Precision	0.88	0.82	**0.94**	**0.94**	-	-	0.71	0.82
	Recall	0.83	**0.88**	0.84	0.80	-	-	0.71	0.74
	F1-Score	0.86	0.85	**0.89**	0.86	-	-	0.71	0.78
Dispersed Coupling	Precision	0.33	0.33	0.78	0.78	-	**0.89**	0.61	-
	Recall	0.75	0.60	0.67	0.67	-	0.62	**0.79**	-
	F1-Score	0.46	0.43	0.72	0.72	-	**0.73**	0.69	-
Feature Envy	Precision	0.50	0.06	**0.94**	0.50	0.81	-	0.75	-
	Recall	0.50	0.50	**0.58**	0.42	0.50	-	0.55	-
	F1-Score	0.50	0.11	**0.71**	0.46	0.62	-	0.63	-
Intensive Coupling	Precision	0.79	0.53	0.63	**0.84**	-	0.74	0.63	-
	Recall	**0.83**	0.71	0.63	0.73	-	0.61	0.67	-
	F1-Score	**0.81**	0.61	0.63	0.78	-	0.67	0.65	-
Large Class	Precision	0.80	0.50	**0.95**	0.85	0.70	0.60	-	-
	Recall	0.80	0.83	0.83	0.85	0.61	**0.92**	-	-
	F1-Score	0.80	0.63	**0.88**	0.85	0.65	0.63	-	-
Long Method	Precision	0.52	**0.80**	**0.80**	0.52	0.76	-	0.72	-
	Recall	0.93	0.95	0.80	0.93	0.83	-	**1.00**	-
	F1-Score	0.67	**0.87**	0.80	0.67	0.79	-	0.84	-
Long Parameter List	Precision	0.56	0.63	0.88	0.50	-	-	**1.00**	0.25
	Recall	**0.69**	0.59	0.54	0.47	-	-	0.57	0.67
	F1-Score	0.62	0.61	0.67	0.48	-	-	**0.73**	0.36
Refused Bequest	Precision	0.40	0.10	0.20	0.30	-	**0.50**	**0.50**	-
	Recall	0.36	0.33	0.18	0.30	-	**0.42**	0.29	-
	F1-Score	0.38	0.15	0.19	0.30	-	**0.45**	0.37	-
Shotgun Surgery	Precision	0.56	0.69	0.50	0.00	-	**0.75**	0.44	-
	Recall	**0.69**	0.58	0.67	0.00	-	0.63	0.50	-
	F1-Score	0.62	0.63	0.57	0.00	-	**0.69**	0.47	-

A. Effectiveness of LLMs in Code Smell Detection (RQ1)

Table VI (first 6 columns) presents the effectiveness in terms of Precision, Recall, and F1-Score of each LLM—DeepSeek-R1, GPT-5 mini, Llama-3.3, and Qwen2.5-Coder—across the code smells analyzed in this study. We highlight in **bold** the best automated strategy, either an LLM or a tool, for the respective metric and smell. Overall, our results indicate that LLMs achieve the strongest and most consistent performance across all metrics for structurally simpler smells, namely *Data Class, Large Class, and Long Method*. For instance, except for GPT-5 mini, which has a Precision of 0.50, the other three LLMs excel at identifying *Large Class* with high Precision: 0.95 for Llama-3.3, 0.85 for Qwen2.5-Coder, and 0.80 for DeepSeek-R1. Similarly, all models achieved a Recall above 0.80 for these three smells, successfully retrieving a broad set of relevant instances. With respect to F1-scores, all LLMs are also highly effective with values above 0.8, with just a few exceptions.

For a second group of three code smells, *Dispersed Coupling, Feature Envy, and Intensive Coupling*, some LLMs present good results while others fail. For instance, Llama-3.3 and Qwen2.5-Coder achieve high Precision (0.78 each) and F1-Score (0.72 each) for *Dispersed Coupling*, while DeepSeek-R1 and GPT-5 mini show substantial drops in effectiveness (e.g., F1 below 0.5), suggesting that these models are not well-suited for detecting this code smell. We can draw similar observations for the other two smells in this group, although the best results are achieved with different LLMs. That is, Llama-3.3 yields better results for *Feature Envy*, and DeepSeek-R1 yields better results for *Intensive Coupling*. In summary, LLM performance is more variable for these three code smells. Therefore, we should carefully select the appropriate LLM that fits our purpose.

When it comes to the most subjective and context-dependent code smells, such as *Long Parameter List, Refused Bequest, and Shotgun Surgery*, all models face substantial challenges. Apart from Llama-3.3 with a Precision of 0.88 for *Long Parameter List*, no model has achieved 0.70 or higher on any metric for these three smells. For *Long Parameter List*, Llama-3.3 achieves relatively high Precision (0.88) but lower Recall (0.54), yielding an F1-score of 0.67. DeepSeek-R1, on the other hand, has a Recall of 0.69, but a low Precision of 0.56. However, the poor effectiveness of LLMs is especially noticeable for *Refused Bequest*, where Precision and Recall both decline, leading all models to score below 0.40 on F1-score. The inherent subjectivity and semantic complexity of these smells, where code context rather than static metrics guides detection, pose significant limitations for LLM-based detections. For instance, *Shotgun Surgery* often involves code scattered across multiple modules, a nuance that requires deep, holistic reasoning that current LLMs rarely replicate.

RQ1 Findings: Overall, the results show that smells based on clear patterns, such as size, class structure, and strong connections between classes, are easier for LLMs to detect. In contrast, smells that require a deeper understanding of the code's context are more challenging for these models. These findings imply that while LLMs can reliably identify certain code smells, there is still work to be done in helping them detect more subjective and complex issues in code.

B. Comparison of LLMs with Static Analysis Tools (RQ2)

Table VI (last 4 columns) provides a comparison of LLMs and traditional static analysis tools for the nine analyzed code smells in terms of Precision, Recall, and F1-Score. Our results show that LLMs often outperform static analysis tools for at least three code smells: *Data Class, Feature Envy, and Intensive Coupling*. For *Data Class*, all LLMs are more effective than static analysis tools across the three metrics, indicating that LLMs are clearly the best choice for this smell. In the case of *Feature Envy*, Llama-3.3 fares best with a notably higher Precision (0.94) and F1-score (0.71). However, other LLMs perform poorly on this smell, revealing inconsistencies across models. For *Intensive Coupling*, JSpIRIT achieves an F1-score of 0.67, which is higher than two LLMs (GPT-5 and Llama-3.3). However, DeepSeek-R1 and Qwen2.5-Coder are better options, suggesting that these LLMs are effective at identifying code smells related to intensive use of method calls. That is, DeepSeek-R1 outperforms both JSpIRIT and Organic in terms of F1-score (0.81) and Recall (0.83), while Qwen2.5-Coder achieves the highest Precision (0.84).

For well-defined code smells, such as *Large Class and Long Method*, both LLMs and static analysis tools often achieve strong results. In fact, the best-performing LLMs tend to outperform or closely match the tools for these smells. For instance, Llama-3.3 achieves the highest Precision (0.95) and F1-score (0.88) for *Large Class*, surpassing JDeodorant and JSpIRIT. However, JSpIRIT outperforms all other strategies in Recall (0.92). Results for *Long Method* follow a similar pattern, but GPT-5 mini shows the highest Precision (0.80) and F1-score (0.87), while Organic achieves perfect Recall (1.00). These results confirm the effectiveness of LLMs in detecting size-related code smells in par with traditional tools.

In contrast, for code smells that are less structurally explicit or more context-dependent, such as *Dispersed Coupling, Long Parameter List, Refused Bequest, and Shotgun Surgery*, the results show greater variation, and traditional static analysis tools often retain an advantage. For instance, JSpIRIT achieves high Precision (0.89) and an F1-score of 0.73 for *Dispersed Coupling*, outperforming all LLMs. Similarly, the Organic tool is the best strategy for *Long Parameter List*, with high Precision (1.00) and F1-score (0.73). On the other hand, *Refused Bequest* and *Shotgun Surgery* remain particularly challenging for both LLMs and static analysis tools. JSpIRIT achieves the highest F1-scores for these two smells (0.45 and 0.69, respectively), far above those of all LLMs. This result shows that while LLMs are expanding the boundaries of automated code analysis, specialized static analysis tools still provide more effective detection for nuanced or scattered patterns that require deeper code context or semantic reasoning.

RQ2 Findings: The results show that LLMs tend to fare better than static analysis tools in detecting simpler and size-based smells. While they are more effective than these tools for five out of the nine smells in this study, traditional tools remain competitive for more subtle or semantically complex code smells. This finding highlights the complementary nature of both strategies and the potential to leverage LLM strengths alongside the established benefits of static analysis tools.

C. A Combined Prediction Strategy for Code Smells (RQ3)

Table VII directly compares, for each code smell, the effectiveness of the best individual strategy with that of the combined prediction strategy proposed in Section IV-C. We consider the best strategy, either an LLM or a static analysis tool, that achieved the highest F1-Score in Table VI. This comparison aims to answer RQ3, that is, to assess the effectiveness of a detection strategy that combines outputs from LLMs and tools. It may also provide valuable insights into which strategy a researcher or practitioner might prefer, depending on their specific detection priorities.

For three code smells, namely *Dispersed Coupling, Large Class, and Long Method*, the combination of multiple tools and LLMs produces the best F1-scores. A common characteristic of this group of smells is that they are related to localized structures and size anti-patterns. For instance, *Large Class* and *Long Method* both achieve their highest F1-scores with the combined prediction (0.93 and 0.89, respectively), compared to their best individual strategies (0.88 and 0.87, respectively). In these cases, the combined strategy delivers the best balance between Recall and Precision, detecting almost all true instances (1.00 for *Large Class* and 0.84 for *Long Method*) while still maintaining high Precision (0.87 for *Large Class* and 0.95 for *Long Method*). For these simpler, localized smells, the combined prediction strategy stands out as the most effective detection method, making it an attractive choice for researchers and practitioners seeking comprehensive, robust results.

Two analyzed code smells – *Data Class* and *Shotgun Surgery* – do not show a clear advantage of the combined prediction strategy over the best individual strategy. Both approaches yield equivalent F1 Scores for these smells (0.89 for *Data Class* and 0.69 for *Shotgun Surgery*), indicating that these strategies produce consistent results. However, it is important to note that the researcher or practitioner would need to know in advance which individual-detection strategy is best. Therefore, a combined strategy may be a more robust option for these two code smells.

In contrast, four code smells, *Feature Envy, Intensive Coupling, Long Parameter List, and Refused Bequest*, are still better detected by a single top-performing tool or LLM. Interestingly, however, LLMs are the most effective for *Feature Envy*

(Llama-3.3) and Intensive Coupling (DeepSeek-R1) while traditional tools fare best for *Long Parameter List* (Organic) and *Refused Bequest* (JSpIRIT). For instance, *Feature Envy* achieves much higher F1-score (0.71) and Precision (0.94) with Llama-3.3 than with the combined strategy (0.60 and 0.50). On the other hand, *Long Parameter List* is best detected by a single tool (Organic) that achieves perfect Precision (1.00) and the highest F1-score (0.73 versus 0.68 with the combined strategy). *Intensive Coupling* and *Refused Bequest* also fit this group, in which the best individual strategies achieve higher F1-scores (0.81 and 0.45, respectively) than the combined strategy (0.77 and 0.30). For these four code smells, sticking with the most effective single strategy provides more targeted and effective detection.

Finally, it is important to note that the combined prediction strategy achieves the best Recall across seven out of nine code smells. Therefore, it seems a rational choice for researchers or practitioners seeking many true instances at the expense of lower Precision. This result is especially valuable for those who prioritize automatic smell detection followed by manual validation. However, it can also lead to a higher rate of false positives and thus higher validation cost for some complex or subjective smells.

TABLE VII
COMPARISON OF THE BEST STRATEGY AND COMBINED PREDICTION

	Best Strategy			Combined Prediction		
Code Smell	**P**	**R**	**F1**	**P**	**R**	**F1**
Data Class	**0.94**	0.84	**0.89**	0.81	**1.00**	**0.89**
Dispersed Coupling	**0.89**	0.79	0.62	0.73	**0.89**	**0.80**
Feature Envy	**0.94**	0.58	**0.71**	0.50	**0.75**	0.60
Intensive Coupling	**0.79**	0.83	**0.81**	0.68	**0.89**	0.77
Large Class	**0.95**	0.83	0.88	0.87	**1.00**	**0.93**
Long Method	0.80	**0.95**	0.87	**0.95**	0.84	**0.89**
Long Parameter List	**1.00**	0.57	**0.73**	0.56	**0.88**	0.68
Refused Bequest	**0.50**	**0.42**	**0.45**	0.30	0.30	0.30
Shotgun Surgery	**0.75**	0.63	**0.69**	0.69	**0.69**	**0.69**

RQ3 Findings: A direct comparison with the best individual strategy shows that the combined prediction works best for code smells with localized structures, such as *Large Class* and *Long Method*, achieving the highest F1-scores and Recall. For more subjective or context-dependent smells, such as *Feature Envy* and *Refused Bequest*, the best individual strategy yields better results. However, when the optimal strategy is unknown, the combined strategy offers a robust alternative. Finally, the optimal strategy depends on whether Recall or Precision is the main priority. In the former case, LLMs are a better option for seven code smells.

VI. DISCUSSION

This section discusses the main findings of this study, the possible uses of the dataset created, and the implications for software practitioners and researchers.

A. Main Findings

Table VIII presents a summary of the effectiveness of each static analysis tool, LLM, and the combined prediction for all code smells analyzed in this study. We rely on the F1-score for each strategy and smell to directly compare their effectiveness. To help interpret the results, we use emojis and a color scale: green indicates high performance (F1-score between 0.80 and 1.00), yellow denotes moderate effectiveness (0.51–0.79), and red signals limited performance (0.50 or below). A check mark indicates the best F1-score for each smell, with ties indicated when multiple strategies share the top F1-score. This table can serve as a practical guide for researchers and practitioners looking to select the most effective detection strategy for each specific code smell.

Several other findings emerged from our study. For instance, one finding relates to code smells with localized structural characteristics, such as *Dispersed Coupling*, *Large Class*, and *Long Method*. For these smells, the combined prediction strategy based on majority voting achieved the highest F1 Scores and Recall compared to both the best single LLMs and tools. For instance, both Large Class and Long Method reached near-perfect Recall and high Precision when leveraging a combined approach. These results show that, for smells directly related to size or clear structural patterns, integrating multiple detection sources helps minimize missed cases and produces robust, highly effective results.

Our evaluation did not identify a clear advantage for either detection strategy for *Data Class* and *Shotgun Surgery*. Both the combined prediction and the best individual strategy (Llama-3.3 and JSpIRIT, respectively) yielded equivalent F1 Scores. This tied effectiveness means developers and researchers can select either strategy based on their workflow or preferences. However, the combined strategy does not require determining the best individual one in advance and thus offers a practical alternative.

Our findings are more nuanced for smells such as *Feature Envy*, *Intensive Coupling*, *Long Parameter List*, and *Refused Bequest*. For these smells, a specialized tool or an LLM often produced the best results, rather than the combined-prediction strategy. For instance, Llama-3.3 performed best at detecting Feature Envy, achieving much higher Precision and F1-score than the combined predictions. Similarly, Organic outperformed all other strategies for Long Parameter List, achieving perfect Precision. These results suggest that, for more subjective or semantically complex smells, specialization and explicit knowledge of inter-class coupling, responsibility allocation, and inheritance hierarchies still outperform the combined prediction strategy, which can sometimes lead to conflicting predictions and reduce overall accuracy.

When comparing LLMs and static analysis tools, our results indicate that LLMs have matured enough to match or exceed traditional tools for six out of nine smells, especially those grounded in clear metrics or modular class designs, such as *Data Class*, *Feature Envy*, and *Intensive Coupling*. However, they continue to struggle where deeper contextual or semantic understanding is necessary. This finding is more evident for *Long Parameter List*, *Refused Bequest*, and *Shotgun Surgery*,

TABLE VIII
EFFECTIVENESS METRICS FOR ALL STRATEGIES WITH TOP RESULTS

	Large Language Models				Static Analysis Tools				Combined Predictions
Smell	**DeepSeek**	**GPT**	**Llama**	**Qwen**	**JDeodorant**	**JSpIRIT**	**Organic**	**PMD**	
Data Class	😐	😐	🏅😊	😐			😐	😐	🏅😊
Dispersed Coupling	☹️	☹️	😐	😐		😐	😐		🏅😊
Feature Envy	☹️	☹️	🏅😐	☹️	😐		😐		😐
Intensive Coupling	🏅😊	😐	😐	😐		😐	😐		😐
Large Class	😊	😐	🏅😊	😊	😐	😐			🏅😊
Long Method	😐	🏅😊	😊	😐	😊		😊		🏅😊
Long Parameter List	😐	😐	😐	☹️			🏅😐	☹️	😐
Refused Bequest	☹️	☹️	☹️	☹️		🏅☹️	☹️		☹️
Shotgun Surgery	😐	😐	😐	☹️		🏅😐	😐		🏅😐

Legend: 😊 (F1 $\geq$ 0.80) 😐 (0.51 $\leq$ F1 $<$ 0.80) ☹️ (F1 $\leq$ 0.50) 🏅 Best strategies

which remain challenging for both LLMs and static tools.

An interesting trend is the systematic improvement of Recall when multiple strategies are combined in a single detection strategy. The combined prediction strategy consistently increased the number of true positive detections, which can be highly beneficial in contexts where missing code smells are a greater risk than occasionally signaling false positives. This finding highlights the strength of the combined prediction strategy as a safer option, especially in automated workflows followed by human review.

Finally, we found that model effectiveness varied not only across code smells but also across LLMs. Llama-3.3 and DeepSeek-R1 often delivered the most effective results among LLMs, suggesting that differences in the architecture of the LLMs and training data can have a measurable impact on their effectiveness for code smell detection. This observation highlights the importance of model selection and the potential value in continuing to fine-tune or customize LLMs for code smell analysis.

B. Expected Use of Our Dataset

Our dataset, built with LLM-detected and human-validated code smells, offers several promising avenues for both researchers and practitioners. By providing not only the raw source code but also detailed detection outputs from multiple LLMs, static analysis tools, and human votes for each class, the dataset becomes a valuable resource for comparison, benchmarking, and the development of new detection strategies.

For researchers, the dataset offers a unique opportunity to investigate the effectiveness of different code smell detectors. As shown in our results, some smells, such as *Large Class* and *Long Method*, can be reliably detected using a combination of automated strategies, whereas others, such as *Feature Envy* and *Refused Bequest*, remain challenging to detect. The performance per-smell of the dataset can be used to motivate and evaluate improvements in smell detection models, the development of more refined LLM prompting strategies, agent architectures, or approaches that incorporate richer code semantics and project context. Researchers can also leverage the chain-of-thought rationale captured in the LLM prompts and outputs to create new detection models.

For practitioners and tool developers, the dataset offers a practical reference for verifying and tuning static analysis tools, LLM-integrated coding assistants, or IDE extensions. Developers working on large codebases can benefit from the dataset's examples of both successful and problematic detections, using them as benchmarks for their own tools or as test cases for CI/CD pipelines. The evaluation provided by human evaluators, established as ground truth, also provides a gold standard against which new tooling can be evaluated and calibrated.

Notably, the structure of our dataset allows for a straightforward extension. Future releases could expand the range of code smells covered, add more languages beyond Java, or include more recent or larger LLMs as they become available. With well-documented prompt templates and evaluation scales, researchers can easily add their own experiments, swap in new models, or repeat our process with code from different domains or repositories. Similarly, collecting additional human votes or involving more experienced or diverse software developers could enrich the reliability and depth of the ground truth.

Another prospective use for our dataset is in educational settings. Instructors can use the varied annotated examples to illustrate standard code smells, help students understand the nuances of different detection methods, or train students in both automated and manual code review practices. The inclusion of multiple perspectives in the dataset (humans, LLMs, and static analysis tools) makes it an excellent resource for critical thinking and for teaching best practices in software engineering.

C. Implications for Researchers

Our findings open several avenues for future research in AI-driven code smell detection. One of the most interesting research opportunities lies in bridging the performance gap for subjective and context-dependent code smells. Despite the increasing capabilities of LLMs and combined detection in structurally explicit smells, such as *Large Class* and *Long Method*, complex cases, such as *Feature Envy* and *Refused Bequest*, remain persistently challenging. Researchers could focus on enriching model context, investigating hybrid models that blend code analysis with design documentation, or building smarter prompting strategies that push LLMs to better understand intent, rationale, and domain-specific patterns beyond structural metrics.

Furthermore, our results reveal the inherent limitations of relying solely on combined predictions for nuanced smells. They also raise questions about the best way to integrate or weight conflicting signals from heterogeneous detectors in a combined or ensemble strategy. Future research might explore adaptive weighting strategies or estimation frameworks that leverage not only the final predictions, but also the reasoning steps provided as "chain-of-thought" explanations. Comparing the robustness and transparency of such hybrid approaches with classical voting could be an important next step.

Another promising direction is advancing explainable AI in code analysis. Our dataset provides detailed "chain-of-thought" reasoning tied to each LLM evaluation. Therefore, researchers can mine these rationales, analyze patterns in LLM logic, and correlate the explanations generated by the models with the correctness of code-smell detections. Insights from the LLM responses could help create a path towards models that not only flag code quality issues but also justify their suggestions in a way developers can trust.

Finally, our work encourages the exploration of transfer learning and cross-domain evaluation. Since our detection pipeline and prompt methodology are open and fully described, researchers could adapt and apply them to new languages and frameworks. By comparing how models perform across different codebases or programming languages, researchers can better understand the extent of AI models' code knowledge and identify areas where adaptation to specific domains is needed.

D. Implications for Practitioners

Our findings provide clear, actionable guidance for practitioners seeking to integrate automated code smell detection into their development processes. One of the most notable practical takeaways is how the effectiveness of a detection strategy depends on the specific type of smell being targeted. For code smells with well-defined, easily quantifiable structures, such as *Large Class* and *Long Method*, practitioners can confidently rely on a prediction strategy that combines several LLMs and tools. These strategies deliver consistently high Recall and F1-score, ensuring that even rare or subtle instances are less likely to escape detection. In legacy systems or software where safety is crucial, high coverage could be prioritized. In these cases, combining detection strategies can help mitigate the risk of smelly code remaining undetected.

However, for more contextual or nuanced code smells, such as *Dispersed Coupling*, *Feature Envy*, and *Long Parameter List*, our results stress the importance of appropriate tool selection. That is, our study shows that a single, specialized tool or well-chosen LLM, such as Llama-3.3 for Feature Envy or Organic for Long Parameter List, offers more accurate, consistent, and interpretable results than collective voting. Selecting the right tool for such smells helps developers avoid "alert fatigue" from false positives and can streamline improvement efforts, saving teams both time and cognitive load.

Practitioners should also weigh the trade-off between missing true smells (false negatives) and generating excessive warnings (false positives). Our results indicate that combined predictions, while strong in Recall, can generate more candidate issues, potentially increasing the cost of manual review. For teams with limited bandwidth or where the developer's trust in automated tooling is paramount, focusing on high-Precision single-method detection for more complex smells is likely to deliver more value. However, projects prioritizing early warning and quality gates may gain greater assurance from combined predictions, even if some follow-up evaluation is required.

Finally, the usability of our dataset extends beyond running benchmarks, enabling practitioners to experiment, calibrate, and continuously improve their chosen strategies. By referencing detailed records of where and why each strategy succeeded or failed, development teams can tune static analysis rules, customize LLM prompt designs, or even blend tool and LLM predictions in custom-made workflows. The traceability and transparency of our results support informed decision-making and promote a learning loop for continuous quality improvement.

VII. Threats to Validity

This section discusses the main validity concerns in our study, including internal, external, and construct validity, as well as reliability. We describe potential threats in each category and outline the steps taken to mitigate them.

Internal validity refers to factors that could affect the results of our study without our knowledge [56]. An important threat is the sensitivity of LLM prompts and configurations, as small changes can yield different results. This feature can make it difficult to attribute the outcomes solely to the model's capabilities. To mitigate these threats, we used the same prompt design and temperature across all LLMs, reported all configuration details, and included all prompts in the replication package to ensure the findings are robust and reproducible.

Another threat to internal validity arises from the eventual bias introduced by the code samples used during LLM training, potentially affecting their ability to detect code smells in previously unseen code. We mitigated this threat by using a dataset containing source code extracted from the 30 most starred GitHub repositories. By selecting widely known code

samples, we ensure that all models are equally likely to have been exposed to the same code during training. This approach promotes consistency in the evaluation.

External validity relates to the extent to which our findings can be generalized beyond the specific context of our study [56]. Generalization to other languages and contexts is a threat to external validity, as results based solely on Java projects may not apply to other programming languages or software domains. Although we understand the limitations of our results, we made it possible to adapt the prompts used in this study to other languages and models since all prompts and scripts are available in our replication package [28].

Construct validity threats arise from how our results are established [56], such as the use of a voting system for the automated prediction strategy and the selection of efficiency metrics. These decisions can introduce bias into the observations. To mitigate them, we grounded our methodology in established practices and recommendations from peer-reviewed studies, ensuring alignment with widely accepted approaches in the field [53], [57]–[59].

Reliability refers to how consistently our findings can be reproduced. If another group of researchers wants to repeat this study under similar conditions, the results need to be comparable. However, variability in responses from models such as GPT-5 mini, Llama-3.3, Qwen2.5-Coder, and DeepSeek-R1, as well as the subjective nature of human detection, could affect reliability. To minimize these threats, we standardized our evaluation procedures, applied the same dataset to all models, and thoroughly documented our methodology. These decisions allow future researchers to replicate our process and achieve similar outcomes. Furthermore, the reproducibility of our results depends on the exact versions of the models and the dataset used. To support reproducibility, we specified the versions of all models used in this study and made the dataset used in our experiments available, thereby providing a clear reference point for future studies.

VIII. Related Work

Researchers are extensively exploring the use of LLMs to assist developers in several software engineering tasks, including code generation [41], software migration [60]–[62], program repair [63], code review [64], and test case generation [65]–[67]. In this section, we provide an overview of studies on LLMs for software development and focus on their use for code smell detection.

Static analysis tools for code smell detection. Prior research has extensively studied code smell detection using static analysis tools [19], [37], [40]. Fernandes et al. [37] conducted a systematic literature review and comparative study of code smell detection tools, revealing substantial overlap and redundancy among tools, as well as wide variation in recall and precision across smells. Paiva et al. [40] evaluated the accuracy and agreement of three popular tools, Deodorant, PMD, and JSpIRIT, showing that tool effectiveness strongly depends on the smell type and that high agreement often stems from identifying non-smelly entities rather than true positives. Tsantalis et al. [19] introduced JDeodorant, a refactoring-aware detection strategy that highlights the benefits of coupling code smell detection with concrete refactoring opportunities. Building upon these foundations, our study investigates whether modern LLMs can overcome some of the limitations observed in traditional tools and earlier machine learning models by leveraging richer contextual understanding of source code.

LLMs have been largely used in software development. Recent studies have explored the strengths and limitations of LLMs in code generation and related software engineering tasks [12], [13], [25], [41], [68]. For example, Liu et al. [25] systematically assessed ChatGPT's code-generation capabilities, highlighting both its strengths and persistent vulnerabilities. Dong et al. [68] proposed a self-collaboration framework in which multiple LLM agents, each with a specific software role, collaborate to improve code generation, achieving notable gains over single-agent approaches. Caumartin et al. [41] showed that, with proper tuning, open-source Llama models can approach ChatGPT's performance in code refinement tasks. O'Brien et al. [13] found that prompt engineering with TODO comments can either help or hinder Copilot's ability to address technical debt, depending on the clarity of the comments. Al Madi [12] found that Copilot's generated code tends to be as readable as human code, but noted that programmers inspect it less, raising concerns about automation bias. Our work complements these efforts by focusing on detecting code smells with LLMs.

LLMs for code smell detection. Some preliminary studies have also investigated how LLMs can be applied to detect and mitigate code smells in different settings [18], [21], [22]. For instance, Silva et al. [21] examined ChatGPT's performance at identifying four classic code smells and found that explicitly naming the smells in the prompt improved detection accuracy, though difficulties persisted in more complex cases. Wu et al. [18] introduced iSMELL, an ensemble technique that integrates LLMs with traditional code analysis tools, outperforming both individual LLMs and single expert systems in detecting and correcting specific code smells. Jiang et al. [22] investigated gas-wasting code smells in Ethereum smart contracts, employing GPT-4 to detect inefficiencies and recommend fixes that led to substantial cost reductions. In contrast, our study examines a broader range of Java code smells, compares proprietary and open-source LLMs, and provides a dataset that combines automated predictions and a human-validated ground truth.

LLMs for code refactoring. In addition to code smell detection, early research explored how LLMs can support code smell refactoring [16], [69]–[71]. For instance, Choi et al. [16] introduced an iterative approach in which ChatGPT 3.5 repeatedly refactors the most complex methods, leading to a steady drop in overall complexity. Pomian et al. [70] developed EM-Assist, an automated strategy that uses ChatGPT 3.5 to suggest and rank "Extract Method" refactorings, achieving a 53% recall for complex cases. Shirafuji et al. [71] presented a strategy to select optimal few-shot examples to guide ChatGPT 3.5 in reducing "Cyclomatic Complexity". More recently, Nunes et al. [69] empirically evaluated two LLMs, namely Copilot Chat and Llama 3.1, for their ability to automatically

fix real-world maintainability issues in Java projects. Their results show that while LLMs can improve code readability and address a subset of code smells, they often introduce compilation errors or new maintainability issues. Unlike these studies, we do not focus on refactoring in this paper. However, our study can be seen as a previous step for improving code by first detecting (our focus) and then refactoring code smells.

IX. Conclusion and Future Work

This study investigated how effective LLMs are at identifying a broad set of nine code smells in 30 Java software projects. In general, our results demonstrated that LLMs, when backed by careful prompt engineering, effectively detect code smells with clear structural patterns, such as those related to class size or complexity. Moreover, we also show that LLMs combined with traditional static analysis tools often match or surpass these strategies in isolation. However, for more subjective or context-dependent smells, such as Feature Envy and Refused Bequest, specialized tools or individual LLMs remain more effective.

In future work, we plan to expand the set of analyzed LLMs by incorporating additional models, such as Sonnet, Claude, and Copilot, and to increase the number of code smells in our dataset. With a robust set of human-validated examples already available in our dataset, we highlight the potential for further research by using these high-quality instances to fine-tune LLMs and explore more advanced prompting techniques, such as few-shot learning. Ultimately, these improvements may yield a more robust assessment of LLMs' potential as reliable tools for automated code-smell detection in real-world software engineering contexts.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 8, pp. 1144–1156, 2012.

[3] A. Uchôa, C. Barbosa, D. Coutinho, W. Oizumi, W. K. Assunçao, S. R. Vergilio, J. A. Pereira, A. Oliveira, and A. Garcia, "Predicting design impactful changes in modern code review: A large-scale empirical study," in *International Conference on Mining Software Repositories (MSR)*, 2021, pp. 471–482.

[4] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 181–190.

[5] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *IEEE International Conference on Software Maintenance (ICSME)*, 2014, pp. 101–110.

[6] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software (JSS)*, vol. 86, no. 10, pp. 2639–2653, 2013.

[7] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 10, pp. 951–976, 2017.

[8] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," in *International Conference on Software Engineering (ICSE)*, 2018, pp. 482–482.

[9] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance (ICSME)*, 2010, pp. 1–10.

[10] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change and fault-proneness," *Empirical Software Engineering (EMSE)*, vol. 17, pp. 243–275, 2012.

[11] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–39, 2014.

[12] N. Al Madi, "How readable is model-generated code? examining readability and visual inspection of github copilot," in *International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–5.

[13] D. OBrien, S. Biswas, S. M. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, "Are prompt engineering and todo comments friends or foes? an evaluation on github copilot," in *International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[14] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," in *International Conference on Software Engineering (ICSE)*, 2023, pp. 2149–2160.

[15] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–6.

[16] J. Choi, G. An, and S. Yoo, "Iterative refactoring of real-world open-source programs with large language models," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2024, pp. 49–55.

[17] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering (TSE)*, 2024.

[18] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "ismell: Assembling llms with expert toolsets for code smell detection and refactoring," in *International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1345–1357.

[19] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 329–331.

[20] PMD, "Pmd source code analyzer," 2025, https://pmd.github.io/.

[21] L. L. Silva, J. R. d. Silva, J. E. Montandon, M. Andrade, and M. T. Valente, "Detecting code smells using chatgpt: Initial insights," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2024, pp. 400–406.

[22] J. Jiang, Z. Li, H. Qin, M. Jiang, X. Luo, X. Wu, H. Wang, Y. Tang, C. Qian, and T. Chen, "Unearthing gas-wasting code smells in smart contracts with large language models," *IEEE Transactions on Software Engineering (TSE)*, 2024.

[23] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *International Conference on Software Engineering (ICSE)*, 2022, pp. 2291–2302.

[24] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software (JSS)*, vol. 203, p. 111734, 2023.

[25] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," *IEEE Transactions on Software Engineering (TSE)*, vol. 50, no. 6, pp. 1548–1584, 2024.

[26] H. G. Nunes, A. Santana, E. Figueiredo, and H. Costa, "Tuning code smell prediction models: A replication study," in *International Conference on Program Comprehension (ICPC)*, 2024, pp. 316–327.

[27] A. Santana, E. Figueiredo, and J. A. Pereira, "Unraveling the impact of code smell agglomerations on code stability," in *IEEE International Conference on Software Maintenance (ICSME)*, 2024, pp. 461–473.

[28] S. Souza, A. Santana, E. Figueiredo, I. Muzetti, J. E. Montandon, and L. Briand, "Replication package of beyond strict rules: Assessing the effectiveness of large language models for code smell detection," 2025, https://github.com/sssouza/code_smell_detection_with_llms.

[29] G. Santos, A. Santana, G. Vale, and E. Figueiredo, "Yet another model! a study on model's similarities for defect and code smells," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2023, pp. 282–305.

[30] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[31] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," *ACM Sigplan Notices*, vol. 34, no. 10, pp. 47–56, 1999.

[32] L. Madeyski and T. Lewowski, "Detecting code smells using industry-relevant data," *Information and Software Technology (IST)*, vol. 155, p. 107112, 2023.

[33] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *International Conference on Software Engineering (ICSE)*, 2011, pp. 1037–1039.

[34] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[35] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: an empirical study," in *International Conference on Technical Debt (TechDebt)*, 2020, pp. 31–40.

[36] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.

[37] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2016, pp. 1–12.

[38] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Working Conference on Reverse Engineering (WCRE)*, Oct 2013, pp. 242–251.

[39] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *Asia Pacific Software Engineering Conference (APSEC)*, Dec. 2010, pp. 336–345.

[40] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development (JSERD)*, vol. 5, pp. 1–28, 2017.

[41] G. Caumartin, Q. Qin, S. Chatragadda, J. Panjrolia, H. Li, and D. E. Costa, "Exploring the Potential of Llama Models in Automated Code Refinement: A Replication Study," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2025, pp. 681–692.

[42] C. Dong, Y. Jiang, Y. Zhang, Y. Zhang, and L. Hui, "Chatgpt-based test generation for refactoring engines enhanced by feature analysis on examples," in *International Conference on Software Engineering (ICSE)*, 2025, pp. 746–746.

[43] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 33, no. 8, pp. 1–79, 2024.

[44] M. H. Tanzil, J. Y. Khan, and G. Uddin, "Chatgpt incorrectness detection in software reviews," in *International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12.

[45] Y. Xue, H. Chen, G. R. Bai, R. Tairas, and Y. Huang, "Does chatgpt help with introductory programming? an experiment of students using chatgpt in cs1," in *International Conference on Software Engineering: Software Engineering Education and Training (CSEE&T)*, 2024, pp. 331–341.

[46] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[47] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering (PACMSE)*, vol. 1, pp. 1703–1726, 2024.

[48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems (NIPS)*, vol. 35, pp. 24 824–24 837, 2022.

[49] A. Bhave and R. Sinha, "Deep multimodal architecture for detection of long parameter list and switch statements using distilbert," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 116–120.

[50] Z. Ságodi, I. Kolláth, P. Hegedűs, and R. Ferenc, "A program synthesis dataset for llm temperature analysis," *IEEE Access*, 2025.

[51] J. Zhang, C.-a. Sun, H. Liu, and S. Dong, "Can large language models discover metamorphic relations? a large-scale empirical study," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2025, pp. 24–35.

[52] S. Ye, Z. Sun, G. Wang, L. Guo, Q. Liang, Z. Li, and Y. Liu, "Prompt alchemy: Automatic prompt refinement for enhancing code generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 51, no. 9, pp. 2472–2493, 2025.

[53] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *European Conference on Information Retrieval (ECIR)*, 2005, pp. 345–359.

[54] W. I. D. Mining, "Data mining: Concepts and techniques," *Morgan Kaufinann*, vol. 10, no. 559-569, p. 4, 2006.

[55] L. Madeyski and T. Lewowski, "Mlcq: Industry-relevant code smell data set," in *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2020, pp. 342–347.

[56] C. Wohlin, *Experimentation in software engineering*. Springer, 2012.

[57] J. Pérez, J. Díaz, J. Garcia-Martin, and B. Tabuenca, "Systematic literature reviews in software engineering—enhancement of the study selection process using cohen's kappa statistic," *Journal of Systems and Software (JSS)*, vol. 168, p. 110657, 2020.

[58] H. Aljamaan, "Voting heterogeneous ensemble for code smell detection," in *IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021, pp. 897–902.

[59] J. P. d. Reis, F. B. e. Abreu, and G. d. F. Carneiro, "Crowdsmelling: A preliminary study on using collective knowledge in code smells detection," *Empirical Software Engineering (EMSE)*, vol. 27, no. 3, p. 69, 2022.

[60] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, "Migrating Code At Scale With LLMs At Google," in *International Conference on the Foundations of Software Engineering (FSE)*, 2025, pp. 1–12.

[61] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, and X. Peng, "LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion," in *International Conference on Software Engineering (ICSE)*, 2025, pp. 781–781.

[62] A. Almeida, L. Xavier, and M. T. Valente, "Automatic Library Migration Using Large Language Models: First Results," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2024, pp. 1–7.

[63] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback," in *ACM International Conference on AI-Powered Software (AIWare)*, 2024, pp. 103–111.

[64] O. B. Sghaier, M. Weyssow, and H. Sahraoui, "Harnessing Large Language Models for Curated Code Reviews," in *International Conference on Mining Software Repositories (MSR)*, 2025, pp. 187–198.

[65] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A Framework for LLM-Based Test Generation," in *International Conference on the Foundations of Software Engineering (FSE)*, 2024.

[66] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *International Conference on the Foundations of Software Engineering (FSE)*, 2024, pp. 185–196.

[67] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 50, no. 1, pp. 85–105, 2023.

[68] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 33, no. 7, pp. 1–38, 2024.

[69] H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Esteves, "Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2025, pp. 669–680.

[70] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "Em-assist: Safe automated extractmethod refactoring with llms," in *International Conference on the Foundations of Software Engineering (FSE)*, 2024, pp. 582–586.

[71] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *Asia Pacific Software Engineering Conference (APSEC)*, 2023, pp. 151–160.