An Exploratory Study on Code Smells Detection and Refactoring using LLMs

Giorgia Paisi
University of Milano-Bicocca
Milano, Italy
g.paisi@campus.unimib.it

Francesca Arcelli Fontana
University of Milano-Bicocca
Milano, Italy
francesca.arcelli@unimib.it

Bartosz Walter
Poznań University. of Technology
Poznań, Poland
bartosz.walter@cs.put.poznan.pl

***Abstract*—With the observed progress in machine learning (ML), and particularly the introduction of Large Language Models (LLMs), several activities related to code maintenance could be automated. That includes not only detection and evaluation of design flaws, but also code transformation and refactoring. However, the general-purpose LLMs, while being commonly used and popular, have not been specifically trained for code analysis, and may not be suitable for conducting software maintenance tasks due to biases, and inherent shortcomings of the models. In this paper, we explore if the widely available LLMs could aid the detection and the refactoring of code smells. We focus on four common smells (God Class, Long Method, Feature Envy, and Refused Bequest) and consider five prompts of diverse complexity, asking the model for detecting and removing the identified code smells. Results suggest that general-purpose LLMs cannot be reliably used for that. They can effectively detect or remove code smells only in simple cases, and frequently produce invalid code. However, their performance depends on various factors, e.g., the model, the specific code smell or the prompt objective and composition.**

***Index Terms*—code smells refactoring, code smells detection, LLM**

I. Introduction

Code smells are surface indicators of possible code or design problems in software systems [1]. Compared to code metrics, they provide a more abstract, but also more holistic view on maintainability: while metrics deliver narrowly-focused, objective values that require interpretation [2], code smells deliver a comprehensive description of the issue and its visible high-level manifestations, making their interpretations much more direct [3]. That way, code smells fill the gap between metric values and comprehensive code analysis [4].

Various approaches to detecting code smells have been proposed, starting from programmers' intuition and experience [1], through quantifiable expressions based on selected metrics [5], [6], analysis of syntactic or dependency graphs [7], tracing the history of code change [8], machine learning-based techniques [9], up to exploiting the knowledge of common relationships among smells [10]. In response, various smell detecting tools have been proposed; however, their convergence remains relatively low [11]. The advent of Large Language Models (LLMs) paved the way for novel techniques in code processing and analysis [12]. Despite several drawbacks regarding contextual awareness and reliability [13], LLMs offer an interesting alternative to the classical approaches to transforming various software artifacts [14], [15].

The goal of this study is to explore whether general-purpose LLMs could be reliably used for code smell analysis. Specifically, we assess their capabilities in both detecting and refactoring four distinct code smells: God Class, Long Method, Feature Envy, and Refused Bequest. By evaluating these dual capabilities, this study reveals whether general-purpose models are effective for practical, end-to-end applications in software maintenance.

II. Related Work

Despite the broad availability of tools, code smells are often left unresolved: according to Shetty et al. [16], over 15% of the detected smells remain unchanged.

LLMs in Code Smell Detection: LLMs' capabilities in code smell detection have recently been assessed by Silva et al. [12]. They found that ChatGPT's performance varies depending on the prompt composition: while detailed prompts contribute to the correct detection, the overall accuracy remains low. In a similar study, Sadik and Govind [17] compared GPT-4.0 and DeepSeek-V3, to find that while GPT-4.0 consistently generated fewer false positives, both models exhibited relatively low recall.

General-Purpose LLM-Driven Refactoring: Recent works studied the use of general-purpose LLMs for conductinf the maintenance tasks. Midolo and Tramontana [18] evaluated ChatGPT's effectiveness in refactoring Java *for* loops into stream pipelines. While it can successfully refactor several cases, its performance is inconsistent, e.g., it struggles with complex loops that require deep contextual understanding. Liu et al. [19] assessed ChatGPT's and Gemini's ability to identify refactoring opportunities and suggest refactoring solutions. ChatGPT produced refactoring solutions that were comparable to or better than the original developers' code in 63.6% of cases, while Gemini achieved this in 56.2% of cases. Karabiyik and Abdulhamid [20] adopted a different approach, introducing RefactorGPT, a ChatGPT-augmented multi-agent

Manuscript received April 29, 2026; revised July 22, 2026; accepted August 07, 2026; published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.145

framework that coordinates four specialised agents, each responsible for a specific task in the refactoring process: Analyzer, Refactor, Refine, and Fixer. This layered design allows a controlled execution and stepwise debugging of the refactoring process.

Fine-Tuned and Domain-Specific LLM-Driven Refactoring: Alazba et al. designed and implemented SmellyBot, a detection tool [21] based on a transformer model pre-trained on large-scale datasets. SmellyBot reported high precision for Data Class, God Class, Feature Envy, and Long Method smells. To support the fine-tuning of LLM and traditional machine learning techniques, Alomari et al. [22] developed a high-quality dataset of code smells with multi-label classification. Wu et al. [15] proposed iSMELL, a MoE (Mixture of Expert) architecture for code smell detection that integrates various specialised toolsets and leverages LLMs to refactor identified code smells. Their evaluation demonstrated that iSMELL outperforms other baselines in detection accuracy and satisfactorily refactors code smells while improving overall code quality attributes. Cordeiro et al. [14] compared refactorings made by humans with those performed by a StarCoder2 LLM-based tool. They reported that the LLM reduced the number of code smells by 44.36%, i.e., 20.1% more than human developers. On the other hand, the LLM-generated code achieved only a 57.15% test pass rate. Moreover, Cordeiro et al. [13] highlighted the limitations of LLM-generated refactorings. They noted that integrating LLMs into IDEs remains difficult due to several factors: limited contextual awareness, a lack of deep understanding regarding the refactoring process, the frequent unavailability of test cases to verify the logic of generated code, and persisting hallucinations.

Although prior work identifies LLMs as promising instruments for code smell detection and refactoring, the results are still fragmented. Many existing studies assess only a narrow subset of code smells and rely on relatively simple prompting approaches. There is no comprehensive investigation of how specific prompt engineering techniques systematically interact with different model architectures across a broad spectrum of code-smell categories; in particular, it is unclear how diverse prompting strategies perform when combined with varying architectures on different types of code smells. This study addresses that gap by conducting a multidimensional analysis that yields a fine-grained understanding of how model-specific capabilities can be exploited to tackle diverse refactoring challenges.

III. Study design

We aim to answer two research questions, focused on the detection and refactoring of code smells.

RQ1: Are LLMs capable of *detecting* code smells?

RQ2: Are LLMs capable of *refactoring* code smells?

The research questions aim to evaluate the effectiveness of LLMs in the code smell-guided maintenance activities. We examine their ability to distinguish between smelly and non-smelly code instances, specifically investigating whether models tend to perform unnecessary refactorings on smell-free code. Furthermore, we analyze how their performance is affected by the specific type of code smell, the prompt engineering technique employed, and the underlying model architecture.

A. Code smells and datasets

We focus on four code smells: God Class (GC), Long Method (LM), Feature Envy (FE), and Refused Bequest (RB), introduced by Fowler [1]. These smells have been extensively studied and documented in the literature [4], [11], [23], and various approaches to the detection and refactoring them have been proposed [24]–[27]. The subject smells capture various violations of coding principles, e.g., concerning cohesion, coupling, and inheritance, at the class and method level. Additionally, they are reliably detected by several static analysis tools, which makes them practical choices for empirical investigation.

To identify a suitable dataset of projects, we followed recommendations by Zakeri et al. [28], and then narrowed down the selection to projects with validated code smell instances of GC, LM, FE, and RB, located in publicly available Java projects with available test suites. As a result, we chose a dataset provided by Cruz et al. [29][1],with 252k non-smelly instances, and 14,918 smelly ones, detected with four code smell detectors: Organic[2] (for GC, FE, and RB smells), JSpIRIT [5][3] (for GC, FE, and RB), JDeodorant[4] (for GC, LM, and FE), and Designite[5] (only for LM). While the original dataset exhibits a significant class imbalance (252,000 non-smelly vs. 14,918 smelly instances), we implemented a controlled sampling strategy to mitigate the potential bias. Specifically, we performed a stratified analysis of approximately 1,300 instances per code smell (GC, LM, FE, and RB), maintaining a balanced 1:1 ratio between the smelly and non-smelly cases to avoid a distorting precision and recall by the majority class distribution.

B. Models, prompts used and evaluation methods

To ensure diversity of data, we reviewed various LLMs, focusing on the popular, general-purpose ones which offered sufficient context sizes at the time of designing the study (March 2025). Based on that, we decided to use four models: *Deepseek-chat*, *Gemini 2.0 Lite*, *Gemini 2.0 Flash*, and *Gemini 1.5 Pro*. Various prompting patterns for LLMs are in common use. In this study, we use five *refactoring-oriented* prompts (R1-R5) and one *detection-oriented* prompt (D1). This selection aims to evaluate the trade-off between the minimalist guidance and structured heuristics. We aim to analyze each model's response and determine whether there is a relationship between the use of a specific pattern and the effects of refactoring.

The prompts are described below.

[1]https://github.com/DVSCross/BadSmellsDetectionStudy
[2]https://github.com/diegocedrim/organic
[3]https://github.com/hcvazquez/JSpIRIT
[4]https://users.encs.concordia.ca/~nikolaos/jdeodorant/
[5]https://github.com/tushartushar/DesigniteJava

Detection-oriented prompt:

- **D1**: A *zero-shot* detection prompt describing the code smell.

Refactoring-oriented prompts:

- **R1**: A *Zero-shot* prompt. It describes the specific code smell and its usual refactorings.
- **R2**: A *One-shot* prompt. It describes the specific code smell, its usual refactorings, and provides a single example of the refactoring strategies.
- **R3**: A *chain-of-thought (COT)* prompt. It includes a step-by-step process to guide the LLM processing.
- **R4**: A *Rule-based* prompt. It contains a series of rules [30] to apply to each possible refactoring.
- **R5**: A *Context* prompt. It only contains a definition of the term "refactoring".

While the D1 prompt focuses solely on the LLMs' detection performance, the refactoring prompts R1-R5 serve the dual purpose of detecting and refactoring the smells. They explicitly instruct the models to apply a refactoring only if the smell is present in the given instance.The verbatim prompts are included in the replication package.

C. Workflow

For each instance in the dataset:

1) Append the instance to six prompts, and send each pair [Prompt, Instance] to all LLMs;
2) Verify if the LLM's response is correctly labeled as *smelly* or *non-smelly*.
3) Only for the refactoring-oriented prompts: 1) Test the LLM-refactored code to verify if it still compiles, and if the tests still pass. 2) Apply the code smell detectors to verify if the smell has been correctly removed.

Evaluation methods: Based on the results, we assigned each instance to one of the following categories:

Resolved: The smell has been removed, and none of the tools detects it anymore.

Enhanced: The smell has been partially removed and is still detected by one or more tools.

Nothing changed: None of the tools detected any difference between the original and the refactored code.

Worsened: The smell is reported by more tools than before.

To classify the instances, we relied on the same smell detectors that were employed to construct the dataset. To reduce the likelihood of circular validation bias, we applied four distinct code smell detectors (as outlined in III-A) and marked an instance as "resolved" only if none of the detectors reported the presence of the smell.

IV. Results

A. Detection performance (RQ1)

To analyse LLMs' detection capabilities, we used one of the detection prompts with both smelly and non-smelly instances. These results were compared against those from the refactoring prompts, which instructed the models to verify the presence of a code smell before attempting a refactoring, to verify whether LLMs prevent unnecessary modification of non-smelly instances. *DeepSeek* failed to identify any smells and, for each instance it responded *"The smell is not present"*. For each model, when using a D1 prompt, as follows from Table I, both *Precision* and *Recall* remained $\leq$0.5.

Model	Precision	Recall	F1
Gemini Lite	0.45	0.17	0.25
Gemini Flash	0.31	0.17	0.22
Gemini Pro	0.50	0.46	0.48

TABLE I
DETECTION PERFORMANCE OF DIFFERENT MODELS (FOR THE D1 PROMPT).

	Detection Prompt			Refactoring Prompt		
Smell	Precision	Recall	F1	Precision	Recall	F1
GC	1.00	0.57	0.72	0.67	0.93	0.78
LM	0.17	0.83	0.28	0.55	0.94	0.69
FE	0.28	0.12	0.16	0.68	0.71	0.69
RB	0.76	0.21	0.33	0.71	0.89	0.79

TABLE II
COMPARISON OF DETECTION PERFORMANCE BETWEEN THE D1 AND R1-R5 PROMPTS

The left side of Table II shows the results for the D1 prompt, organized by code smell. In terms of *Precision*, the detection of *GC* and *RB* performs best, achieving scores of 1 and 0.76, respectively. In contrast, *LM* and *FE* are below 0.30. Regarding *Recall*, it is different: *LM* achieves the best result with 0.83, followed by *GC* at 0.57. Both *FE* and *RB* barely reach 0.20, demonstrating limited *Recall* capabilities. The *F1 score* highlights how *GC* achieves the best overall performance (0.72).

R1-R5 prompts, even though the detection step was only linked to a simple conditional statement, deliver better results. In this case, as shown in the right side of Table II, *F1 scores* for all code smells are between 0.69 and 0.79. *RB* has the best *Precision* (0.71), while *GC* and *FE* show similar values (0.67 and 0.68 respectively). *LM* ranks lowest with 0.55. In terms of *Recall*, *LM* and *GC* reach 0.94 and 0.93, respectively. *RB* follows closely with 0.89, while *FE* is the lowest, at 0.71. These results indicate that the use of a refactoring-oriented prompt improves the LLMs' detection performance and yields more consistent results compared to the detection-specific approach.

Regarding RQ1, we noticed that LLMs struggle in consistently detecting code smells, with negligible differences when using a detection-specific prompt. Better performance is observed when the prompt focuses on a different task (e.g., refactoring) and detection occurs as a side effect. When using a detection-oriented prompt, the ability of LLMs to detect code smells is strongly dependent on the specific type of smell, with *GC* achieving the highest *F1 score*. In contrast, when performing detection with a refactoring-oriented prompt, all code smells have improved *F1 scores*, and the LLMs perform consistently better across different smells. These results are interesting, as refactoring-oriented prompts only contained a

simple conditional statement concerning the smell detection. It is important to note that the study primarily focuses on refactoring of code smells, and only one detection-specific prompt was used. Consequently, the ability of LLMs to discern when refactoring is necessary (and when to refrain) emerges as a contextual strength rather than a simple pattern-matching capability, justifying our multidimensional focus on the interplay between task framing and model architecture.

B. Refactoring performance (RQ2)

Even though the LLMs were asked to refactor both smelly and non-smelly instances, this section focuses on the refactoring results for smelly instances. As reported in Sec. IV-A, R1-R5 prompts yielded considerable detection results, indicating that they rarely attempt to refactor non-smelly code. To analyze the refactoring performance and address **RQ2**, it is important to consider the *build results*: the percentage of code generated by the LLMs that compiles successfully and passes all unit tests in their project's test suite. To analyse DeepSeek's capabilities alongside those of the other models, the explicit detection request was removed from its prompts, while keeping the remainder unchanged. Since any difference between DeepSeek's and Gemini's prompts might affect the comparability, the prompt was only changed only by removing a short statement sentence. This alteration, however, poses a threat to validity of results. Within the Gemini family, the performance differences between models are consistent, showing a progressive improvement that aligns with each model's capabilities. The build success rate is highly dependent on both the code smell and the model, as shown in Table IV: percentages vary for the same model, when refactoring different code smells. Contrarily, the relationship between the chosen prompt and the build success rate appears less evident (Table V): percentages remain similar when the refactoring is executed by the same model. None of the models has an overall build success rate above 50%, and significant debugging skills are required to identify defects in the generated code. To further evaluate the refactoring performance, only the code successfully built was analyzed, and no manual debugging was performed. Each refactoring was categorized as described in the Sec. III-C.

Models	Successful Builds
Gemini Lite	31.11%
Gemini Flash	32.89%
Gemini Pro	42.13%
DeepSeek	17.91%

TABLE III
BUILD SUCCESS RATES IN DIFFERENT MODELS (REFACTORING-ORIENTED PROMPT)

Qualitative analysis: A manual inspection of the LLM-generated code for the failed builds revealed several recurring issues. Models frequently attempted to access private methods from external classes, failing to account for Java's access modifiers. In several instances, the models changed the visibility of original classes from public to private, resulting in compilation failures. Another frequent source of defects was the LLMs' tendency to hallucinate variables and method return types; this led to attempts of casting objects to incompatible types and pass incorrect argument types to constructors and methods. Furthermore, during refactoring, models often renamed methods intended to override a supertype, thereby breaking the inheritance chain and disrupting polymorphic behaviour. Finally, LLMs frequently hallucinated non-existent packages, methods, and variables. These patterns suggest that while the models understand local scope, they struggle to grasp the wider context of the project and often produce low-quality code.

Code Smells	GC	LM	FE	RB
Gemini Lite	25.21%	29.23%	56.31%	20.27%
Gemini Flash	17.36%	27.47%	63.47%	22.91%
Gemini Pro	16.31%	68.73%	42.57%	23.67%
DeepSeek	0.95%	15.46%	46.42%	8.01%

TABLE IV
BUILD SUCCESS RATES IN DIFFERENT MODELS FOR DIFFERENT CODE SMELLS (REFACTORING-ORIENTED PROMPT)

Prompts	0-shot	1-shot	COT	Rule-based	Context
Gemini Lite	25.44%	26.43%	27.08%	31.16%	41.24%
Gemini Flash	31.93%	35.11%	22.53%	30.08%	42.20 %
Gemini Pro	41.45%	38.85%	39.87%	35.97%	50.06%
DeepSeek	16.32%	17.64%	16.23%	16.81%	22.29%

TABLE V
BUILD SUCCESS RATES IN DIFFERENT MODELS FOR DIFFERENT PROMPTS (REFACTORING-ORIENTED PROMPT)

Is there a difference in the refactoring capabilities based on the smell?

Table VI reveals a correlation of the refactoring capability with the specific code smell ($p < 0.001$, Pearson's Chi-square test). When refactoring *LM*, LLMs tend to perform better: in 18.75% of the cases, the code quality is *Enhanced*, and in 44.02% of the cases, the smell is entirely *Resolved*. Only 6.92% of the time the code quality is *Worsened*. This is different for *GC* and *FE*: both show a rate of *Nothing changed* exceeding 50%, indicating that quality has neither increased nor worsened. Even so, in both cases, the percentage of *Solved* is comparable to the percentage of *Worsened* cases (16.04% vs. 1.86% for *GC*, and 13.71% vs. 19.45% for *FE*). *RB* shows the poorest performance among all code smells. Although the successful build rate is comparable to the others, the refactoring produces little to no improvement in the code.

Code smells	GC	LM	FE	RB
Resolved	16.04%	44.02%	13.71%	1.18%
Enhanced	11.19%	18.75%	6.99%	0%
Not. changed	70.89%	30.29%	59.83%	98.09%
Worsened	1.86%	6.92%	19.45%	0.71%

TABLE VI
PERFORMANCE OF REFACTORING SPECIFIC SMELLS WITH ALL PROMPTS

Is there a difference in the refactoring capabilities based on the prompt?

Prompts	0-shot	1-shot	COT	Rule*	Context
Resolved	20.64%	20.14%	25.65%	16.15%	21.06%
Enhanced	10.60%	10.53%	6.79%	9.98%	10.10%
Not. changed	52.65%	53.97%	59.64%	60.07%	60.17%
Worsened	16.09%	15.34%	7.89%	13.79%	8.65%

TABLE VII
PERFORMANCE OF REFACTORING ALL SMELLS WITH SPECIFIC PROMPTS. *Rule: Rule-based*

Table VII reports the refactoring performance for different prompts. It is difficult to identify a clear relationship between the prompt type and refactoring performance. All prompts show the *Nothing Changed* rate between 50% and 60%, with the percentage of *Solved* exceeding that of *Worsened* quality. The only significant differences are related to the COT, which yields the best results ($p < 0.001$, Pearson's Chi-squared test) with the highest *Solved* and lowest *Worsened* rate, and Context, which has the highest *Nothing Changed* rate of 60.17%. The difference appears to be linked to the level of detail in the prompts: the prompts that exhibit the best performance are also those that best explain the smell and its refactoring.

Is there a difference in the refactoring capabilities based on the LLM?

Models	G. Lite	G. Flash	G. Pro	DeepSeek
Resolved	12.26%	17.58%	12.63%	55.37%
Enhanced	7.51%	7.72%	17.89%	7.98%
Not. changed	71.60%	59.23%	53.89%	32.78%
Worsened	8.61%	15.45%	15.57%	3.85%

TABLE VIII
PERFORMANCE OF REFACTORING ALL SMELLS WITH ALL PROMPTS USING SPECIFIC MODELS

Table VIII reports the refactoring performance by model. The data shows that *DeepSeek* outperforms all Gemini models ($p < 0.001$, Pearson's Chi-squared test), which exhibit comparable performance, exceeding their *Resolved* rates by a factor of three. However, these results lack comprehensiveness: *DeepSeek* has a low *Build Success* rate, with most successful compilations occurring for *LM* and *FE*. It is therefore difficult to estimate *DeepSeek*'s true refactorings' efficacy, as these smells appear to be the easiest to refactor and the data are unevenly distributed. The three Gemini models exhibit similar results overall: *Gemini Pro* produces better refactorings, but also a higher share of *worsened*-quality code. The differences among the Gemini models appear to be consistent with their cost and abilities.

A closer inspection of Table IX, reveals another pattern: each model's refactoring performance varies depending on the specific code smell. *DeepSeek* achieves a 96.05% success rate when refactoring *LM* and a 52.24% success rate for *FE*, making it the best-performing model for these two code smells. Conversely, *DeepSeek* shows the worst results for *GC* and *RB*, with a 0% success rate. These data, together with its 17.91% *Build Success* rate, make this model the least suitable for refactoring code smells. In contrast, all three Gemini models exhibit different performance across the various code smells. The *Solved* and *Enhanced* rates (cases when an improvement was detected) of **GC** are consistent with model ranking: *Gemini Lite* has the lowest value (3.63% and 11.81%), *Gemini Flash* falls in the middle (26.42% and 10%), and *Gemini Pro* achieves the highest (15.38% and 23.07%). Both *Nothing Changed* and *Worsened* rates follow similar patterns, decreasing as the model improves. Concerning **LM**, although *Gemini Lite* and *Gemini Flash* follow the trend described above, with 37.55% and 69.27% *Resolved* rates respectively, *Gemini Pro* shows significantly lower performance, with only 17.53% of *Resolved* instances. *Gemini Pro* also exhibits the highest *Worsened* rate. When refactoring *FE* all three models deliver similar results, with comparable *Resolved* and *Nothing Changed* rates. Finally, to analyze LLMs' capabilities to refactor **RB** the results reported in Table VI are fundamental: RB is the least-refactored code smell, with a 100% *Nothing changed* rate for both *Gemini Lite* and *Gemini Pro*, and a 96.33% rate for *Gemini Flash*. Although *Gemini Flash* exhibits better results, only 2.29% of the code smells are *Resolved*.

Smells	GC	LM	FE	RB	GC	LM	FE	RB
Models	**Gemini Lite**				**Gemini Flash**			
Resolved	3.63	37.55	6.23	0	26.42	69.27	5.43	2.29
Enhanced	11.81	19.24	3.23	0	10.0	12.84	8.48	0
Not. changed	82.72	34.27	77.36	100	61.42	17.87	57.55	96.33
Worsened	1.81	8.92	13.16	0	2.14	0	28.52	1.37
	Gemini Pro				**DeepSeek**			
Resolved	15.38	17.53	6.21	0	0	96.05	52.24	0
Enhanced	23.07	26.49	6.21	0	0	3.94	10.61	0
Not. changed	61.53	44.02	63.84	100	100	0	31.42	100
Worsened	0	11.94	23.72	0	0	0	5.71	0

TABLE IX
LLMS' RESULTS (IN PERCENT) OBTAINED REFACTORING SMELLS WITH ALL PROMPTS.

Qualitative analysis: Refactoring code with an LLM involves trade-offs among different types of code smells, as a manual analysis of results from all categories (Solved, Enhanced, Nothing Changed and Worsened) revealed. For instance, refactoring *LM* instances resolves *Complex Method* and *Long Statement* smells, but introduces *Magic Numbers* and *Long Parameter Lists* in return. Similarly, the refactoring of *GC* increases the number of *FE*, suggesting that LLMs struggle to maintain proper encapsulation when decomposing large classes. Moreover, *RB*'s refactoring occasionally resolves *FE* only to introduce *DC* smells.

With respect to **RQ2**, the refactoring capabilities of LLMs depend on the model and the specific code smell. Accuracy is affected by the chosen prompt, with more detailed prompts performing better. As described in Table III, these data must be interpreted in light of the *Build Success* rates: none of the models exceeds 50% in successful builds.

V. Discussion

Our results indicate that the subject models were unable to consistently detect code smells when the detection was their only task. This finding aligns with the works by Silva et al. [12] and Sadik and Govind [17], who reported that LLMs detection performance is heavily prompt-dependent. Similar to Sadik and Govind's observations of GPT-4.0, the Gemini models in our study generally achieved higher precision than recall (Table I), indicating a conservative tendency to overlook code smells. However, when using refactoring-oriented prompts, the result reversed, and recall increased significantly. Notably, the F1-score improved alongside precision, signaling that refactoring-oriented instructions help the model identify smells more accurately without increasing false positives.

We achieved a moderate *successful builds* rate between 17% and 42%. This is notable, as Cordeiro et al. [14] reported a first-pass performance of 28.4% using a StarCoder2, a model specifically trained for code generation. Our results demonstrate that general-purpose LLMs could achieve refactoring capabilities comparable to those of models trained for code generation. As described in Sec. IV-B, model performance depends on the specific code smell. The best results were obtained for *LM*, while *RB* was the most challenging. Cordeiro et al. [14] observed similar results, reporting better performance for syntactic refactoring tasks compared to more complex refactoring requiring broader contextual understanding. Furthermore, this mirrors Midolo and Tramontana's [18] observation that LLMs struggled with transformations requiring deeper contextual understanding.

Furthermore, our findings regarding prompt engineering reinforce the consensus in recent literature; similar to Tessa et al. [31] and Pandini et al. [32], who found that detailed prompts boost performance for architectural smells, we identified *Chain-of-Thought* as the most effective strategy. This confirms that providing the model with a logical path to follow is more effective than simply providing Context, which we found to be the least robust.

Threats to validity The collected results may not reflect all real-world scenarios, which constitutes a threat to *external validity*. As demonstrated, the LLM refactoring ability varies significantly across specific code smells; therefore, reported results cannot be extrapolated to other smell types. Furthermore, model performance is highly sensitive to the prompting strategy. While we mitigated this by using a set of six different prompt types, these factors inherently limit the extent to which the results represent the models' absolute capabilities. Our decision to alter DeepSeek's prompt to generate outputs may have also introduced bias into the results. Moreover, depending on external code smell detectors for both building the dataset and assessing the LLM-generated code may have introduced a circular validation bias, which we attempted to mitigate by using multiple tools. Also the choice of a specific LLM also affects the accuracy for both csmell detection and refactoring. The analyzed set of LLMs, prompting strategies, and code smells was significantly smaller as compared to all possible scenarios. Therefore, the results could not be directly transferred and generalized. However, they demonstrate trends and limitations that are likely to be present in the similiar settings as well.

Another factor is related to the fact that we used the subject LLMs' as autonomous refactoring agents rather than interactive tools. This differs from the "human-in-the-loop" or "multi-agent" approaches analyzed in recent literature, such as RefactorGPT [20]. Our goal was to test whether general-purpose LLMs could independently identify and refactor code smells, thereby providing assistance to less-experienced developers. Finally, to evaluate models that are commonly available, we limited our analysis to two model families, and this choice may have influenced the data, as highlighted in Sec. IV. Also, StarCoder2 [14] and Code Llama [33], designed explicitly for code generation, might produce different results. To mitigate this issue, each of the four models responded to an average of 1300 [Prompt, Code Smell] random pairs.

Finally, the temporal constraints allowed for running each prompt only once, which does not take into account the stochastic nature of LLMs.

VI. Conclusions

In this study we presented how four commonly available, general-purpose LLMs performed in code smell detection and refactoring of Java programs.

There are two main contributions of the work. First, the subject LLMs detected code smells more accurately if they were explicitly instructed to refactor the code, and the detection was only an intermediate step towards the objective. The effect was better when combined with a Chain-of-Thought prompt, which mandated the model to plan and explain the steps undertaken to complete the task. On the other hand, the prompt composition had only minimal impact on the LLMs' refactoring performance: while **prompts containing more information tend to produce better results**, this advantage was not significant. The second finding is that the model performance varied per type of code smell. Code smells that could be resolved with simpler refactorings, such as LM, achieved better results. In contrast, **refactorings that required a deeper understanding of the code frequently failed and introduced new code smells as a side effect.**. A similar observation concerns the choice of the model: while none of them seemed to clearly dominate others, some of them consistently provided incorrect responses.

Results indicate that the analyzed general-purpose LLMs, without being fine-tuned for code analysis, are not capable of refactoring code smells without human oversight. **They frequently produce code that does not compile and only rarely apply refactorings correctly. Hence, we recommend using such models only as supportive tools to assist informed developers, rather than as independent, unsupervised agents capable of substituting software engineers in refactoring code smells.**

The findings of this study lay the ground for further research in several directions. To address the problem of low

success rates in complex refactorings and the hallucinations observed in this study, future work could incorporate Retrieval-Augmented Generation (RAG) and specialized, refactoring-focused LLMs into the pipeline. This could help to determine whether narrowly specialized training could mitigate the identified risks and the performance gap. Beyond autonomous performance, the *cost of correction* warrants investigation. It remains an open question whether the time required for a developer to fix the code generated by a model outweighs the speed of manual refactoring. Furthermore, the impact likely varies based on developer seniority; therefore, further studies should discriminate between novice and expert developers' outcomes to identify where LLMs' assistance is most viable. Given the stochastic nature of LLMs, the reliability of a *single-pass* approach represents a significant methodological risk. Future experimental designs should explore whether multiple iterations or *self-debugging* cycles, where the model receives feedback on the compile errors, could resolve the issues identified in this study, or if models tend to converge on the same defects regardless of the number of attempts.

The replication package is available in Zenodo[6].

References

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

[2] 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, 2006.

[3] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*. IEEE, 2003, pp. 381–384.

[4] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 350–359.

[5] S. Vidal, H. Vazquez, A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," 2015.

[6] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[7] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *2007 IEEE International Conference on Software Maintenance*, 2007.

[8] 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 *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[9] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4

[10] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, 2017, pp. 8–13.

[11] B. Walter, F. A. Fontana, and V. Ferme, "Code smells and their collocations: A large-scale experiment on open-source systems," *J. Syst. Softw.*, vol. 144, pp. 1–21, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2018.05.057

[12] L. L. Silva, J. R. d. Silva, J. E. Montandon, M. Andrade, and M. T. Valente, "Detecting code smells using chatgpt: Initial insights," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024.

[13] J. Cordeiro, S. Noei, and Y. Zou, "Llm-driven code refactoring: Opportunities and limitations," *2025 IEEE/ACM Second IDE Workshop (IDE)*, pp. 32–36, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:277677973

[14] ——, "An empirical study on the code refactoring capability of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2411.02320

[15] 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 *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1345–1357. [Online]. Available: https://doi.org/10.1145/3691620.3695508

[16] G. Shetty and T. Sharma, "Mapping code smells and refactorings accurately: Insights from an empirical study," Preprint, 2025. [Online]. Available: https://tusharma.in/preprints/ESEM2025_Smells_Refactoring_Mapping.pdf

[17] A. R. Sadik and S. Govind, "Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3," 2025. [Online]. Available: https://arxiv.org/abs/2504.16027

[18] A. Midolo and E. Tramontana, "Refactoring loops in the era of llms: A comprehensive study," *Future Internet*, vol. 17, no. 9, 2025.

[19] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "An empirical study on the potential of llms in automated software refactoring," 2024. [Online]. Available: https://arxiv.org/abs/2411.04444

[20] M. A. Karabiyik, "Refactorgpt: a ChatGPT-based multi-agent framework for automated code refactoring," *PeerJ Computer Science*, vol. 11, 2025. [Online]. Available: https://doi.org/10.7717/peerj-cs.3257

[21] A. Alazba, H. Aljamaan, and M. Alshayeb, "Smellybot: An ai-powered software bot for code smell detection," *Software: Practice and Experience*, 2025.

[22] N. Alomari, A. Alazba, H. Aljamaan, and M. Alshayeb, "Smellycode++: Multi-label dataset for code smell detection," *Scientific Data*, 2025.

[23] N. Anquetil, A. Etien, G. Andreo, and S. Ducasse, "Decomposing god classes at siemens," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[24] A. Kovacevic, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokic, and G. Sladic, 2022.

[25] M. Shahidi, M. Ashtiani, and M. Zakeri, "An automated extract method refactoring approach to correct the long method code smell," *Journal of Systems and Software*, 2022.

[26] D. Yu, Y. Xu, L. Weng, J. Chen, X. Chen, and Q. Yang, "Efficient feature envy detection and refactoring based on graph neural network," *Automated Software Engineering*, vol. 32, 12 2024.

[27] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," 09 2013.

[28] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, 2023.

[29] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: an empirical study," 2020.

[30] K. Prete, N. Rachatasumrit, and M. Kim, "Catalogue of template refactoring rules," *Univ. of Texas at Austin, Tech. Rep. UTAUSTINECE-TR-041610*, 2010.

[31] C. Tessa, M. Bochicchio, and F. Arcelli Fontana, *Exploring Architectural Smells Detection Through LLMs*, ser. Lecture Notes in Computer Science. Springer, 2025, vol. 15929, pp. 90–98. [Online]. Available: https://doi.org/10.1007/978-3-032-02138-0_6

[32] G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana, "An exploratory study on architectural smell refactoring using large languages models," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, 2025.

[33] B. Rozière, Gehring, Gloeckle, Sootla, Gat, Tan, Adi, Liu, Remez, J. Rapin, Kozhevnikov, I. Evtimov, Bitton, Bhatt, Ferrer, A. Grattafiori, Xiong, D'efossez, Copet, Azhar, Touvron, Martin, N. Usunier, Scialom, and Synnaeve, "Code llama: Open foundation models for code," *ArXiv*, vol. abs/2308.12950, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261100919

[6]https://doi.org/10.5281/zenodo.17572881