Journal of Software Engineering Research and Development, 2026, 6:1, doi: 10.5753/jserd.2026.7984
This work is licensed under a Creative Commons Attribution 4.0 International License.

LLMs on code smell detection and refactoring: a systematic literature review

Mateus Dias [**Universidade Federal de Viçosa** | ***mateus.j.dias@ufv.br***]
Lucas Vegi [**Universidade Federal de Viçosa** | ***lucas.vegi@ufv.br***]

Abstract

Large Language Models (LLMs) are increasingly being applied to software maintenance tasks such as code smell detection and code refactoring, yet the evidence on their use remains fragmented across heterogeneous tasks, evaluation settings, and tool architectures. In particular, there is still limited consolidated knowledge about the prompt-based use of frozen-parameter LLMs for these activities. This paper presents a systematic literature review of how such models have been applied and evaluated in code smell detection and refactoring. Following a predefined protocol, we combined database searches and snowballing and used source-grounded AI assistance during screening under human oversight, yielding a final corpus of 65 studies. The review shows that this is a recent and rapidly expanding research area, concentrated around a small number of model families, programming languages, and canonical smells and refactorings. Across the reviewed studies, standalone prompting was generally less reliable in structurally demanding scenarios, whereas hybrid and repository-aware approaches were more often associated with stronger outcomes. Retrieval-augmented generation, validation layers, and iterative workflows repeatedly emerged as key mechanisms for improving reliability. Beyond technical performance, the evidence also indicates that practical usefulness depends on transparency, iterative interaction, and visible developer control. Overall, this review consolidates a still-dispersed body of work and provides a focused basis for future research and for the design of context-aware, validated, and human-supervised LLM-based maintenance tools.

Keywords: *Large Language Models, Code Smells, Refactoring, SLR*

1 Introduction

Large Language Models (LLMs) are transformer-based foundation models trained on massive corpora of natural language and, in many cases, source code, enabling them to model, generate, and transform textual and code-related content across a wide range of tasks (Li et al., 2025). In recent years, this family of models has expanded to include general-purpose systems, code-oriented variants, instruction-following assistants, and retrieval-augmented configurations, making LLMs increasingly flexible for different forms of computational support (Naveed et al., 2025). In software engineering, their use has rapidly extended beyond code completion to a broad spectrum of activities, including requirements-related support, code generation, testing, repair, documentation, design, and maintenance tasks (Fan et al., 2023). More recently, LLMs have also been explored in software-quality-oriented activities, including code smell detection and code refactoring (Wu et al., 2024), suggesting that these models are increasingly being positioned not only as productivity aids, but also as assistants for reasoning about design quality and behavior-preserving change.

Code smells are commonly understood as symptoms in source code that may indicate deeper design or maintainability problems, whereas refactoring refers to behavior-preserving changes intended to improve the internal structure of software without altering its external functionality (Fowler and Beck, 1999). Long before the recent rise of LLMs, both topics had already become established research areas in software engineering. In the case of code smells, prior review evidence points to a long-standing literature centered on traditional smell categories and multiple detection strategies, including metric-based, rule-based, and other automated approaches, often accompanied by heterogeneous findings and fragmented evaluation settings (Sobrinho et al., 2021). Likewise, refactoring research had already evolved into a broad and methodologically diverse field involving recommendation, detection of refactoring opportunities, automation, validation, and correctness assessment across many software contexts (Abid et al., 2020). Against this background, the use of LLMs for code smell detection and refactoring is still a comparatively recent development, and the emerging literature remains scattered across distinct technical, methodological, and human-centered directions, with no consolidated and widely accepted body of knowledge yet regarding how these models should be applied, evaluated, or integrated into maintenance practice.

Despite the growing number of studies in this area, the literature still lacks a consolidated understanding of what constitutes effective, reliable, and practically useful application of LLMs for code smell detection and refactoring. The emerging body of work remains methodologically fragmented, with studies differing substantially in task framing, prompting strategy, evaluation criteria, datasets, and surrounding tool support, making results difficult to compare directly. This gap is especially relevant because the present review focuses on a specific and practically important subset of the field: the prompt-based use of frozen-parameter LLMs, in which task adaptation is achieved at inference time through prompting and in-context learning rather than through fine-tuning or retraining. Although one recent systematic literature review has focused specifically on LLM-based refactoring (Martinez et al., 2026), and a broader review has examined the use of LLMs for code quality enhancement, includ-

ing refactoring and code smell detection among other tasks (Alomari et al., 2026), there is still no review centered explicitly on the intersection of prompt-based, frozen-parameter LLMs with both code smell detection and code refactoring. To address this gap, the present study conducts a systematic literature review aimed at consolidating the available evidence, identifying recurring patterns and limitations, and deriving insights that can guide future research and tool design in this emerging area.

To achieve this objective, we conducted a Systematic Literature Review (SLR) following established review guidelines and a predefined protocol covering search strategy, eligibility criteria, study selection, and data extraction. The review combined database searches with complementary snowballing procedures and applied explicit inclusion and exclusion criteria to isolate studies on the prompt-based use of frozen-parameter LLMs for code smell detection and refactoring. A distinctive aspect of the methodology was the use of NotebookLM as a source-grounded support mechanism during study screening: rather than replacing human judgment, it was used to provide a second, evidence-linked assessment of candidate papers and to support discrepancy analysis during reconciliation. After selection, the included studies were analyzed through both demographic and technical mapping and a structured qualitative synthesis organized around technical, methodological, educational, and human-centered dimensions.

The main contributions of this paper are fourfold. First, it provides a focused synthesis of the emerging literature on the prompt-based use of frozen-parameter LLMs for code smell detection and code refactoring, a scope that remains insufficiently consolidated in prior review work. Second, it shows that, in technically demanding scenarios, standalone prompting tends to be less reliable than hybrid and repository-aware approaches, especially when tasks depend on project-level structure, inter-class reasoning, or validation against compilation and testing constraints. Third, it identifies recurring workflow patterns associated with more reliable outcomes, including retrieval-augmented generation, task decomposition, static-analysis support, testing-based validation, and feedback-driven regeneration, suggesting that effective LLM-based maintenance support is better understood as an orchestrated pipeline than as a one-shot generation task. Finally, it extends the discussion beyond technical performance by showing that the usefulness of these systems also depends on educational and human factors: LLMs appear more beneficial when used in reflective, guided, and human-supervised settings, while practical adoption remains tied to transparency, iterative interaction, and visible developer control.

The remainder of this paper is organized as follows. Section 2 discusses related work and positions the present study with respect to prior reviews on code smells, refactoring, LLM-based software engineering, and AI-assisted evidence synthesis. Section 3 presents the research methodology, including the review protocol, search strategy, eligibility criteria, study selection process, and data extraction procedure. Section 4 reports the results and discussion, combining a demographic and technical mapping of the included studies with a critical synthesis of the main technical, methodological, educational, and human-centered findings. Section 5 derives practical implications for the design of LLM-based tools for code smell detection and refactoring. Section 6 revisits the findings through a consolidated assessment of the effectiveness, reliability, and practical usefulness of these approaches. Section 7 discusses the threats to validity of the review. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Related Work

Research related to this study spans three partially overlapping lines: **(i) reviews on LLM-based refactoring**, **(ii) broader reviews on the use of LLMs for code quality enhancement**, and **(iii) literature reviews on code smells and refactoring as established software engineering topics**. Taken together, these works provide an important background, but they do not fully address the specific intersection examined here, namely the prompt-based use of frozen-parameter LLMs for both code smell detection and code refactoring.

A first and directly related study is the systematic literature review by Martinez et al. (2026) on software refactoring research with large language models. Their review analyzes 50 primary studies and synthesizes the literature around refactoring methods, prompting techniques, LLM tools, datasets, benchmarks, and recurring challenges, including erroneous code generation and difficulties with more complex refactoring tasks. This makes it the closest review to the present work. However, its scope is centered on refactoring, whereas our review jointly examines code smell detection and refactoring under a narrower prompt-based, frozen-parameter perspective.

A second closely related review is the study by Alomari et al. (2026) on the use of LLMs to enhance code quality. Their review covers 49 studies and explicitly includes both code smell detection and code refactoring, but does so within a broader umbrella of code-quality-related tasks. It is therefore an important adjacent reference for the present study. At the same time, its scope is wider than ours, both thematically and methodologically, since it treats smells and refactoring as part of a larger code quality landscape and includes a broader range of models and adaptation strategies than the prompt-based frozen-parameter focus adopted here.

A broader historical foundation is provided by prior reviews on code smells and refactoring as long-standing software engineering topics. Sobrinho et al. (2021) show that bad-smell research has long concentrated on traditional smell categories, heterogeneous detection strategies, and fragmented empirical settings, helping explain why conclusions in the area are often difficult to compare directly. Abid et al. (2020), in turn, show that refactoring research has evolved into a large and diverse field spanning recommendation, detection of refactoring opportunities, automation, validation, and quality assessment across many contexts. These works are essential for situating the present review historically, but they predate the recent LLM-centered wave and therefore do not address how prompt-based LLMs are being used in smell detection and refactoring workflows.

Overall, the related literature provides substantial knowledge on code smells as a software quality topic, refactoring as a mature research field, and LLM-based refactoring and code-quality enhancement as emerging research directions. What remains insufficiently covered is their intersection. To the best of our knowledge, there is still no review centered specifically on the prompt-based use of frozen-parameter LLMs for both code smell detection and code refactoring, while also integrating technical, methodological, educational, and human-centered evidence within the same synthesis. The present study addresses this gap through a scoped and methodologically explicit review of how this subset of LLM applications has been used and evaluated in the software maintenance literature.

3 Research Methodology

This study was conducted as a Systematic Literature Review (SLR) following the formal guidelines established by Kitchenham and Charters (2007). The methodology was designed to provide a transparent, objective, and reproducible pipeline for mapping the state of the art in prompt-based LLM applications for code smell detection and refactoring, excluding approaches centered on fine-tuning, retraining, or other model adaptation strategies.

A central feature of this research is the adoption of a hybrid selection strategy that combines manual screening with an AI-assisted stage using NotebookLM[1]. In this study, the AI-assisted stage was used as a source-grounded support mechanism for discrepancy analysis and evidence tracing, rather than as a replacement for human judgment. The manual and AI-assisted screenings were conducted separately prior to reconciliation, and final inclusion decisions remained under the responsibility of the first author.

Figure 1 summarizes the steps undertaken to conduct this SLR. These steps are detailed in the following subsections.

3.1 Planning and Protocol Definition

The planning phase of this SLR involved the definition of a formal protocol to ensure objectivity and minimize researcher bias. This protocol established the Research Questions (RQs) and the criteria for study selection.

3.1.1 Research Questions

The study is driven by two primary Research Questions designed to investigate how prompt-based LLMs with frozen parameters have been applied and evaluated in code smell detection and refactoring tasks within the software maintenance lifecycle.

> **RQ1:** How have prompt-based LLMs, used without fine-tuning or retraining, been utilized and evaluated for detecting code smells in the literature?
>
> **RQ2:** How have prompt-based LLMs, used without fine-tuning or retraining, been utilized and evaluated for code refactoring in the literature?

RQ1 aims to synthesize how prompt-based, frozen-parameter LLMs have been applied to code smell detection from technical, methodological, and human-centered perspectives. It examines reported evaluation strategies and performance indicators — such as accuracy, precision, recall, and F1-score — across different interaction and architectural settings, including standalone prompting and hybrid workflows supported by static analysis or retrieval mechanisms. In addition, it considers how the reviewed studies discuss developer interaction, trust, and explanatory support in smell-related tasks.

On the other hand, RQ2 investigates how prompt-based, frozen-parameter LLMs have been applied to code refactoring workflows within the scope of this review. It examines how refactoring effectiveness has been evaluated, including criteria such as compilation success, test-passing behavior, structural validity, and preservation of intended behavior. It also considers how developers interact with these systems in refactoring scenarios, with attention to human-in-the-loop validation, workflow orchestration, and collaborative use in software maintenance settings.

3.1.2 Inclusion and Exclusion Criteria (IC/EC)

To ensure the selection of high-quality primary studies relevant to the RQs, we defined a set of Inclusion Criteria (IC) and Exclusion Criteria (EC). A study was selected for the final review only if it met all ICs and did not trigger any ECs.

In addition to topical and methodological eligibility, publication characteristics were also considered during study assessment. The review primarily targeted scholarly contributions disseminated through established scientific databases and digital libraries, including peer-reviewed conference and journal publications. However, publication venue or peer-review status was not used as an absolute exclusion criterion. Individual scholarly contributions, including book chapters and preprints identified through the complementary snowballing process, remained eligible when they satisfied the substantive inclusion criteria and could be assessed as independent research contributions. Publication type and venue were subsequently recorded during data extraction to characterize the composition of the evidence base.

Inclusion Criteria (IC):

- **IC1 — Direct relevance to LLM-based code smell detection or refactoring:** The study must investigate the application of Generative Large Language Models (LLMs) as active agents for detecting code smells and/or performing refactoring tasks. The LLM must be directly used, analyzed, or evaluated in the context of improving structural code quality.

[1] NotebookLM was renamed Gemini Notebook by Google in July 2026. See: `https://notebooklm.google.com/`

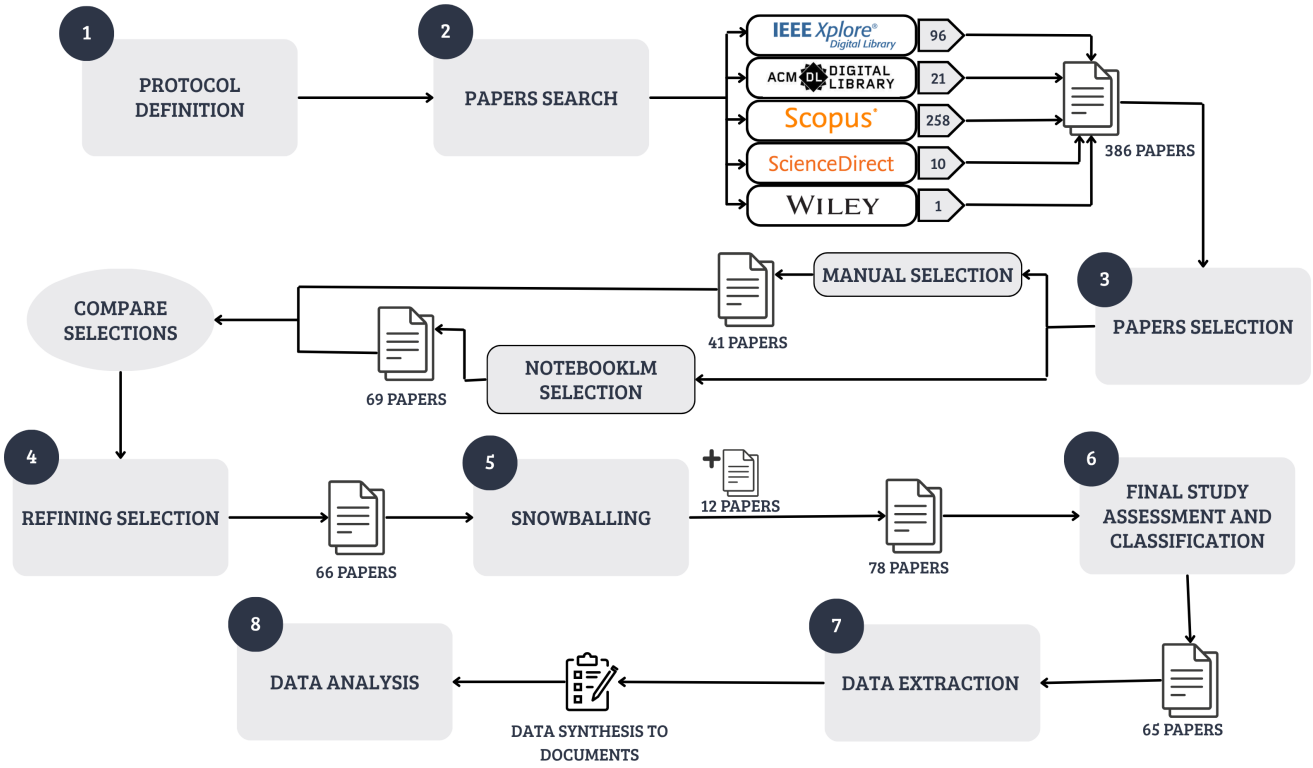


Figure 1. Overview of the SLR process.

- **IC2 — Study providing an original contribution or substantive analysis involving LLMs in the context of code smell detection or refactoring:** The study must present an original methodological proposal, empirical evaluation, architectural discussion, or analytical examination of the use of LLMs for code smell detection or refactoring. Works that merely reference prior studies or outline future research directions without providing a concrete methodological, analytical, or conceptual contribution were excluded.
- **IC3 — Inclusion of general-purpose programming languages:** The study must evaluate or analyze source code written in at least one general-purpose programming language (*e.g.,* Java, Python, C++, or Elixir). Studies focusing exclusively on domain-specific languages or infrastructure-as-code (*e.g.,* SQL, HTML, or Terraform) were excluded. For the purpose of this review, a general-purpose programming language was understood as a language designed to support the development of software across a broad range of application domains, whereas a domain-specific language was considered to be tailored to a particular problem domain or class of applications. This distinction follows established characterizations of DSLs in the software engineering literature (van Deursen et al., 2000).
- **IC4 — Publication period:** The study must have been published or made publicly available online between 2019 and November 5, 2025, the date on which the automated search was conducted.

Exclusion Criteria (EC):

- **EC1 — Focus exclusively on bug fixing or automated program repair:** Studies were excluded if they focus exclusively on bug fixing or automated program repair without addressing structural code quality, code smells, or refactoring.
- **EC2 — Studies based primarily on fine-tuning or retraining LLM weights:** Studies were excluded if their primary method involves fine-tuning, retraining, or modifying model weights, rather than using Generative LLMs as inference-time agents through prompting or related techniques.
- **EC3 — Non-granular publication units:** Records representing complete books, edited volumes, or entire conference proceedings were excluded, as they do not constitute individual primary studies and cannot be assessed as independent units of analysis. Individual book chapters and standalone conference papers were considered eligible when meeting the inclusion criteria.

3.2 Search Strategy and Paper Identification

A systematic automated search was executed on November 5, 2025, across major digital libraries in the software engineering field: IEEE Xplore, ACM Digital Library, ScienceDirect, Scopus, and Wiley. The search string (detailed in Table 1) was designed to capture the intersection of LLMs and code smell detection/code refactoring strategies. The initial search yielded 386 papers, each assigned a unique identifier ranging from P001 to P386 to ensure traceability, as documented in our replication package. Throughout this work, citations to these papers are presented alongside their respective P-prefixed identifiers.

The temporal scope of this review was restricted to studies published from 2019 to 2025. This decision is grounded in the consolidation of Large Language Models (LLMs) as a

Table 1. SLR Query String Specification.

Search Query	`( "large language model" OR "llm" OR "generative ai" OR "transformer" OR "chatgpt" OR "gemini" OR "claude" OR "deep seek" OR "grok" OR "llama" OR "foundation model" OR "codex" OR "copilot" )` **AND** `( "refactoring" OR "refactor" OR "code smell" OR "code smells" OR "bad smell" OR "bad smells" OR "antipattern" OR "antipatterns" OR "anti-pattern" OR "anti-patterns" OR "code improvement" OR "code refinement" OR "program transformation" )`

dominant paradigm in Natural Language Processing (NLP) and its subsequent application to Software Engineering (SE).

While the Transformer architecture proposed by Vaswani et al. (2017) provided the foundational multi-head attention mechanisms, and 2018 saw the introduction of models like GPT-1 (Radford and Narasimhan, 2018) and BERT (Devlin et al., 2019), these early iterations were intermediate-scale models with limitations in breadth and generative capacity.

The year 2019 represents a pivotal turning point with the release of GPT-2 (Radford et al., 2019). Featuring 1.5 billion parameters, GPT-2 was the first large-scale model to exhibit significantly superior performance in complex text generation and comprehension tasks. This release not only coined the term *"Large Language Model"* but also inaugurated a contemporary phase of research characterized by massive datasets and the systematic exploration of these models within the software development lifecycle, including testing and maintenance. Therefore, the 2019 cutoff ensures that this SLR captures the most technologically mature and relevant literature regarding the integration of LLMs in code smell detection and refactoring.

3.3 Study Selection Process

The study selection was conducted in distinct stages to ensure transparency and reduce the risk of prematurely excluding relevant primary studies. The process combined an initial manual screening with a subsequent AI-assisted analysis and a final reconciliation step, with all inclusion decisions ultimately validated by the first author.

3.3.1 Methodological Justification for the Selection Strategy

The selection phase is widely recognized as one of the most critical stages of an SLR, as relevance judgments may be affected by incomplete abstracts, ambiguous terminology, and researcher interpretation (Kitchenham and Charters, 2007). These challenges have motivated increasing interest in the use of LLMs as assistive mechanisms during study screening, particularly as a means of providing an additional source of evidence without replacing human judgment. In software engineering, Felizardo et al. (2024) evaluated ChatGPT as an aid to the study-selection activity and emphasized its role as a supporting opinion rather than a substitute for researcher assessment. More recently, Felizardo et al. (2025) discussed broader methodological challenges associated with conducting and replicating LLM-assisted SLRs, including transparency, prompt design, reproducibility, and accuracy.

These observations informed the methodological stance adopted in the present review. AI assistance was used to support evidence inspection and discrepancy analysis, while final inclusion decisions remained under human responsibility. Following this principle, we integrated NotebookLM as a source-grounded second-opinion mechanism alongside the manual screening conducted by the first author. Its role was to provide traceable textual evidence from the analyzed papers and to support the re-examination of discrepant cases during reconciliation, rather than to automate the final inclusion decision.

The selection of NotebookLM over general-purpose LLMs is grounded in its specialized architecture, which addresses the core challenges of research integrity and evidence-based synthesis in the context of an SLR. Recent investigations into the adoption of LLMs in SLRs highlight significant methodological concerns regarding transparency, replicability, and reliability when using general-purpose models (Felizardo et al., 2025). The methodological justifications for this choice are detailed below:

- **Source-Grounded Structural Integrity:** Unlike general-purpose LLMs that derive answers from vast internal parametric memory — often leading to risks of hallucination or ungrounded synthesis — NotebookLM operates under a source-grounded paradigm. Studies discussing the use of LLMs in SLR contexts point out that hallucinations and lack of grounding pose serious threats to scientific validity, particularly in technical domains such as Software Engineering (Felizardo et al., 2025). By constraining generation strictly to an uploaded and explicitly defined corpus, a source-grounded system reduces epistemic uncertainty and mitigates fabrication risks. Empirical analyses of NotebookLM in academic review contexts further indicate that grounding responses in provided documents improves factual consistency and reduces unsupported claims compared to open-domain LLM usage (Shor et al., 2025).
- **Traceability and "Black Box" Mitigation:** A well-documented limitation of traditional LLMs is their "black box" nature, which complicates justification and auditability of automated decisions (Felizardo et al., 2025). NotebookLM mitigates this limitation by providing explicit, hyperlinked citations to specific passages within the source corpus that support each generated claim. This feature directly addresses one of the key open challenges identified in recent SE research on LLM-supported SLRs: the lack of interpretability and justification mechanisms in automated study selection (Felizardo et al., 2025). By enabling sentence-level traceability, the tool supports human-in-the-loop validation and auditability, strengthening methodological transparency. Evidence from domain-specific evaluations of NotebookLM further suggests that citation-linked outputs enhance researcher trust and facilitate verification workflows during literature synthesis (Shor et al., 2025).

3.3.2 Stage 1: Manual Screening and Deduplication

Following the search phase, duplicate entries were identified and removed, reducing the initial set of 386 results to 279

unique papers. The first author then performed an initial manual screening of these studies. Following the guidelines proposed by Brereton et al. (2007), the title and abstract of all unique papers were analyzed. In cases where the title and abstract provided insufficient information to reach a clear inclusion or exclusion decision, the conclusion section was also consulted (23 cases) to support a more informed preliminary assessment. Each study was evaluated against the predefined inclusion and exclusion criteria in order to determine its relevance to the research questions.

This manual pass was conducted before the AI-assisted phase so that the preliminary judgments would not be influenced by NotebookLM outputs. In cases where the first author considered the eligibility decision uncertain, the study was temporarily marked as MAYBE. These borderline cases were then re-examined by the second author at title-and-abstract level, providing an additional human assessment to support the evaluation of ambiguous cases. The manual and AI-assisted screening stages were therefore carried out separately, allowing AI-generated evidence to be examined only during the later reconciliation stage. This procedure was intended to reduce anchoring effects and to preserve the independence of the initial manual assessment.

3.3.3 Stage 2: AI-Assisted Selection (NotebookLM)

The AI-assisted selection phase required full-text access to leverage NotebookLM's source-grounded analysis. Of the 279 unique papers, 37 were immediately excluded under EC3 (Non-granular publication units) because they corresponded to complete books or full conference proceedings rather than individual primary studies. Of the remaining records, 20 studies could not be retrieved in full via institutional subscriptions or open-access repositories. Due to the logistical infeasibility of acquiring these specific full texts, they were excluded from this stage.

Consequently, the remaining 222 papers were uploaded to NotebookLM for independent analysis. To ensure methodological isolation and prevent cross-document interference, a single study was assigned to each NotebookLM project, resulting in a total of 222 individual notebooks. Although the NotebookLM Standard plan allowed the inclusion of up to 50 sources per project at the time of the study,[2] incorporating multiple documents within the same notebook may require internal retrieval and content selection mechanisms that could introduce contextual noise or reduce response precision. By restricting each notebook to a single source, we ensured that all generated outputs were exclusively grounded in one document, thereby eliminating the risk of contextual dilution or inadvertent information blending across different studies.

To improve the consistency of AI-generated outputs, several pilot prompts were tested and refined before the final screening prompt was defined (see Figure 2). The final prompt was designed using Zero-Shot Instruction Prompting combined with Role Prompting, following the taxonomy presented by Sahoo et al. (2025). The model is explicitly assigned the role of an SLR researcher to align its responses with systematic review standards and methodological rigor. The prompt employs a structured, criteria-driven instruction format that decomposes the screening task into explicit evaluation steps (*i.e.,* method, experiment, objective, and language), guiding analytical reasoning without relying on demonstration examples. Additionally, it enforces structured output prompting by requiring predefined response fields and evidence-based justification through direct textual quotations, ensuring traceability and reproducibility in study selection decisions. The prompt assigned the model the role of an SLR researcher with the specific goal of strictly filtering papers based on four technical dimensions:

- **Methodological Constraint:** The study must evaluate a Generative LLM using Prompt Engineering (*e.g.,* Zero/Few-shot, RAG, Agents). Crucially, the prompt included an explicit exclusion for Fine-Tuning, requiring the model to verify if the LLM was used with frozen parameters.
- **Task Alignment:** The analysis focused on Refactoring and Code Smell identification. The logic was programmed to strictly exclude Bug Fixing and Automated Program Repair (APR), ensuring the review remained centered on structural code quality rather than logic error correction.
- **Experimental Originality:** To distinguish primary studies from secondary ones, the prompt required evidence that the authors performed their own experiments or provided original analysis rather than merely citing related work.
- **Language Scope:** The model was instructed to include any study involving General-Purpose Languages (GPL), while excluding those focusing solely on Domain-Specific Languages (DSLs) or Infrastructure-as-Code (IaC).

Rather than generating a simple binary decision, the model was required to produce a structured analysis for each paper. This format was designed to facilitate human verification and ensure that every verdict was backed by textual evidence. The output followed a mandatory template:

- **Evidence Collection:** The model provided direct quotes from the paper regarding the tool used, the task performed, and the programming language analyzed.
- **Reasoned Verdict:** A final "INCLUDE" or "EXCLUDE" decision accompanied by a brief technical justification.

All generated analyses, including the extracted quotations and justifications, were systematically stored in a spreadsheet. This repository served as the primary input for the reconciliation phase, allowing the first author to inspect the textual basis of each AI suggestion and compare it with the initial manual decision. These files are available in the study's replication package.[3]

3.3.4 Application of Inclusion and Exclusion Criteria

To ensure methodological rigor, transparency, and reproducibility, the Inclusion Criteria (IC) and Exclusion Criteria

[2] `https://support.google.com/notebooklm/answer/16213268`. Accessed: 2026-02-20

[3] `https://doi.org/10.5281/zenodo.19368366`

Prompt for Study Selection

Role: SLR Researcher.
Goal: Strictly filter papers.
Context: Select studies where a Generative LLM acts as the agent for refactoring/detection. We seek code quality improvements, not bug fixing or optimization.

Criteria (CI):
1. Tool and Method: Study evaluates a Generative LLM (*e.g.,* GPT, Llama, Claude) using Prompt Engineering strategies (*e.g.,* Chain-of-Thought, RAG, Agents, Zero/Few-shot).
1.1 Constraint: EXCLUDE Fine-Tuning. The study must use the LLM with frozen parameters. If the primary method is "retraining", "fine-tuning", or "updating weights" (common in CodeT5/BERT), decide EXCLUDE.
2. Task: Investigates the application of LLMs in code smell identification or refactoring execution.
2.1 Constraint: Strictly EXCLUDE Bug Fixing / Automated Program Repair (APR). The goal must be Code Quality/Structure (Refactoring), not fixing logic errors or code optimizations.
2.2 Constraint: The study must perform the experiment or demonstrate it. The text should present results and/or an analysis. If it only cites other papers doing refactoring (*e.g.,* in "Related Work") without its own experiment or analysis, decide EXCLUDE.
3. Target: General-Purpose Languages (GPL) (Java, Python, Elixir, etc.).
3.1 Logic: INCLUDE if the study analyzes ANY GPL (even if mixed with DSL). EXCLUDE only if the study focuses SOLELY on DSLs/IaC (*e.g.,* only Terraform or SQL).

Instructions:
1. Check Method: Is it Prompt-based (Frozen Model)? If Fine-Tuning -> EXCLUDE.
2. Check Experiment: Did the authors run the LLM for refactoring/detection in this paper? (If just citing others -> EXCLUDE).
3. Check Goal: Is it Refactoring/Quality? (Only Bug Fixing/APR -> EXCLUDE).
4. Check Language: Is at least one GPL present? (Only DSL/IaC -> EXCLUDE).

Output Format:
Analysis:
Tool/Method Evidence: [Quote]
Task/Agency Evidence: [Quote proving LLM does the refactoring/detection] - [Reasoning]
Language Evidence: [Quote]
Verdict: [INCLUDE / EXCLUDE] Reason: [Brief justification]

Figure 2. Prompt used in NotebookLM to screen studies in the SLR.

(EC) defined in Section 3.1.2 were applied using a structured hierarchical evaluation protocol, as summarized in Table 2. Each study was assessed sequentially against the criteria in order of priority. Once a study failed to satisfy an inclusion criterion or met an exclusion criterion, the evaluation process was immediately terminated, and the first unmet criterion was recorded as the primary reason for exclusion.

Table 2. Hierarchical Order of Inclusion and Exclusion Criteria Applied During Study Selection.

Priority	Criterion
1	IC1 — Topic relevance
2	IC2 — Original contribution involving LLM application or analysis
3	IC3 — Target programming language scope
4	EC1 — Bug fixing only
5	EC2 — Fine-tuning-based approaches

The publication period constraint (IC4) was operationalized during the database search stage as a retrieval filter; consequently, all records entering the screening phase inherently satisfied this requirement. Additionally, EC3 (Non-granular publication units) was applied prior to the formal screening phase to remove complete books and full conference proceedings, ensuring that only individual primary studies were subjected to hierarchical evaluation.

The remaining criteria were applied uniformly across all screening stages, including the initial manual assessment and the subsequent AI-assisted selection. Table 3 presents a comprehensive quantitative summary of the study selection process, documenting the progressive refinement of the study pool from the initial retrieval to the final set of included primary studies. The table reports the number of studies excluded at each stage, including duplicate removal, unretrievable records, and exclusions resulting from violations of the defined inclusion and exclusion criteria. For studies excluded based on IC or EC violations, the recorded exclusion reason corresponds to the first unmet criterion according to the hierarchical protocol described above.

To further illustrate the practical application of the criteria, representative examples of studies that were excluded—either due to the application of exclusion criteria or the failure to meet inclusion criteria—are presented next, along with supporting textual evidence.

The failure to meet IC1 was the most frequent reason for excluding papers during the study selection phase, accounting for the removal of 97 studies. The quote presented in the next excerpt, referring to paper P148 (Zhu et al., 2025), clearly situates the study within the domain of computer vision, specifically addressing universal image segmentation. Although the terms "Refactored" and "Refactor" appear in the title and abstract, they refer to architectural modifications of a neural network rather than to source code refactoring in the context of software engineering. The study does not analyze source code artifacts, structural quality concerns, code smells, or the use of Generative LLMs as agents for software engineering tasks. Therefore, it does not satisfy Inclusion Criterion IC1.

IC1 Application Example – Quote from Paper P148

> "The introduction of DEtection TRansformers (DETR) has marked a new era for universal image segmentation in computer vision. [...] In this paper, we propose a novel architecture called Refactored Maskformer, which builds upon the Mask2Former through two key modifications: Decoupler and Reconciler."

On the other hand, the failure to meet IC2 was the least frequent reason for excluding papers at this stage of the study, accounting for only five exclusions. The quotation presented in the excerpt below, referring to paper P069 (Sun, 2024), illustrates an example of a study that does not meet IC2. It indicates that the work primarily outlines intended contributions and a proposed framework, without reporting an implemented system, experimental validation, or empirical evaluation. As the study remains at a conceptual or developmental stage without operationalizing or assessing the proposed LLM-based solution, it does not meet the requirement of a finalized primary contribution.

Table 3. Complete Study Selection and Refinement Process.

Phase	Description	Remaining Studies
Initial Retrieval	Records identified through database search	386
Duplicate Removal	Removal of duplicate records (107)	279
Structural Filtering	EC3 — Non-granular publication units (books/proceedings) removed (37)	242
Accessibility Check	Full-text not retrieved (20)	222
Hierarchical Screening Criteria Applied		
Content Screening	IC1 — Topic relevance excluded (97)	125
	IC2 — Original LLM contribution excluded (5)	120
	IC3 — Target programming language scope excluded (6)	114
	EC1 — Bug-fixing only studies excluded (30)	84
	EC2 — Weight fine-tuning or retraining approaches excluded (18)	66
Preliminary Included Studies	Studies eligible after refinement	66
Snowballing Addition	Additional studies identified (+12)	78
Full-Text Analysis	Version consolidation (4)	74
	Removed under EC2 (5)	69
	Removed under IC1 (2)	67
	Removed under IC2 (2)	65
Final Study Set	**Studies included for data extraction**	**65**

IC2 Application Example – Quote from Paper P069

> "By the end of this project, the contributions are expected to be as follows: Enhanced Log Analysis and Reusability [...]. Development of a Real-Time Feedback Framework [...]"

Paper P007 (Vo et al., 2025) was one of the six studies excluded during the selection process due to not meeting IC3. As shown in the quote presented in the excerpt below, the cited passage explicitly states that the study operates solely on Terraform configurations, which constitute an Infrastructure as Code (IaC) language. Because the evaluation is restricted to this domain-specific configuration language and does not involve general-purpose programming languages, the study does not satisfy IC3 and was therefore excluded.

IC3 Application Example – Quote from Paper P007

> "Terraform is a widely used Infrastructure as Code (IaC) tool that simplifies cloud resource management through declarative configuration. [...] In this paper, we harness large language models (LLMs) for automated Terraform code smell detection... We construct a synthetic benchmark dataset of 42 Terraform configurations covering 14 distinct code smells and evaluate detection performance [...]"

Exclusion Criterion EC1 was the most frequently applied among those listed in Table 2, accounting for the removal of 30 papers during the selection process. The statement presented in the excerpt below is a quotation from paper P050 (Jin et al., 2024) that exemplifies the application of this criterion. It demonstrates that the proposed framework focuses on automatic debugging and iterative correction of faulty code through multi-agent LLM collaboration. Although the term *"refinement"* is used, it refers to incremental debugging and correction during code generation rather than to systematic structural refactoring or code smell mitigation. The study does not investigate structural quality attributes, design-level improvements, or refactoring operations. Therefore, because its scope is restricted to debugging and program repair, the study satisfies EC1 and was excluded.

EC1 Application Example – Quote from Paper P050

> "[...] we introduce a novel architecture of LLM-based agents for code generation and automatic debugging: Refinement and Guidance Debugging (RGD). The RGD framework is a multi-LLM-based agent debugger that leverages three distinct LLM agents—Guide Agent, Debug Agent, and Feedback Agent. RGD decomposes the code generation task into multiple steps, ensuring a clearer workflow and enabling iterative code refinement based on self-reflection and feedback."

Finally, the next excerpt presents an example of the application of Exclusion Criterion EC2, which was responsible for the exclusion of 18 papers among the 386 studies originally retrieved in the search process. The excerpt, extracted from paper P008 (Habib et al., 2025), clearly indicates that the study applies pre-trained transformer models to perform supervised multi-label classification of code smells at the class level. Such an approach characteristically involves task-specific fine-tuning on labeled datasets to optimize predictive performance. Therefore, as the primary methodology centers on fine-tuning and supervised retraining, the study satisfies EC2 and was excluded.

EC2 Application Example – Quote from Paper P008

> "This paper examines the use of pre-trained transformer models, RoBERTa, TinyBERT, DistillBERT, ALBERT & DistillRoberta to perform accurate multi label classification of code smells in a class level dataset."

3.3.5 Selection Consolidation and Refinement

After completing the manual and AI-assisted screening phases separately, a consolidation and refinement process was conducted to determine the preliminary set of included studies. This stage aimed to reconcile the two screening outputs while preserving human responsibility for the final judgment.

It is important to clarify that the manual and AI-assisted outputs were not designed as two equivalent annotations produced from the same information source, nor was their comparison intended to measure inter-rater agreement. The initial manual screening constituted a preliminary relevance assessment based primarily on titles and abstracts, whereas the subsequent NotebookLM analysis deliberately used full texts to provide complementary, source-grounded evidence that might not be visible during preliminary screening. The purpose of comparing the two outputs was therefore diagnostic rather than evaluative: disagreements were used to identify studies requiring re-examination, particularly cases in which relevant methodological details emerged only from the full text. Consequently, the reported YES/NO distributions should be interpreted as part of the reconciliation procedure rather than as agreement measurements between equivalent reviewers.

The reconciliation protocol followed two rules. First, any study marked as "YES" in the initial manual screening was retained for the preliminary included set. Second, studies marked as "NO" in the manual screening but as "YES" by NotebookLM were re-examined by the first author through a targeted review of the AI-generated notes, supporting quotations, and the corresponding full-text passages. This step was designed to identify cases in which relevant methodological details were not sufficiently visible in the title, abstract, or conclusion during the initial manual pass.

Across the 222 full-text papers analyzed with NotebookLM, the comparison between the initial manual screening and the NotebookLM screening yielded the following distribution: 29 studies were classified as YES/YES, 141 as NO/NO, 40 as NO/YES, and 12 as YES/NO. After the reconciliation step, the comparison between the final author decision and the NotebookLM screening became 54 YES/YES, 141 NO/NO, 15 NO/YES, and 12 YES/NO. These numbers indicate that NotebookLM was more inclusive than the initial manual pass and was particularly useful for surfacing potentially relevant studies that warranted re-examination. At the same time, the remaining NO/YES cases also show that the AI output was not treated as self-validating and still required human adjudication.

These distributions also explain the intermediate counts shown in Figure 1. The 41 studies retained by the initial manual screening correspond to the 29 YES/YES cases plus the 12 YES/NO cases, whereas the 69 studies suggested for inclusion by NotebookLM correspond to the 29 YES/YES cases plus the 40 NO/YES cases. These numbers therefore represent the outputs of two separate screening stages and should not be interpreted as successive reductions of the same study pool. During reconciliation, the 40 studies classified as NO manually but YES by NotebookLM were re-examined by the first author using the AI-generated evidence and corresponding full-text passages. Of these, 25 were retained and 15 remained excluded. Since the 41 studies initially retained by the manual screening were preserved, this process resulted in the preliminary set of 66 studies (*i.e.*, 41 + 25) reported after the refinement stage in Figure 1. The screening counts summarized in Table 3 refer to the final reconciled eligibility decisions and their recorded exclusion reasons, rather than to the NotebookLM output alone.

A recurrent source of disagreement involved the interpretation of the term refactoring. In several cases, NotebookLM marked papers as potentially relevant because they used terms such as *"refactoring"*, *"transformation"*, or similar restructuring language. However, after full-text verification, some of these studies were excluded because they addressed tasks such as code translation, platform migration, or cross-language conversion, which do not fit the stricter definition of refactoring adopted in this review, namely, the behavior-preserving restructuring of existing code. This pattern suggests that the AI-assisted screening was useful for sensitivity, but also prone to semantic over-inclusion in conceptually adjacent tasks.

At the end of this reconciliation stage, a preliminary set of 66 studies was obtained for the subsequent snowballing step. As reported later in Section 3.5.1, this set should not be interpreted as the final evidence corpus for extraction, since a further full-text eligibility verification was still performed after snowballing.

3.4 Automated Backward Snowballing Process

To ensure the comprehensiveness of our study, we implemented a multi-stage automated pipeline for Backward Snowballing. This process was developed based on the recommendations of Wohlin (2014) for conducting manual snowballing in systematic reviews of software engineering studies, with the aim of identifying relevant studies cited by the primary set of papers selected in the previous phase.

Figure 3 expands the sub-steps implemented in Step 5 of the SLR process presented in Figure 1. The following paragraphs provide a detailed description of the snowballing procedure employed in this work.

1. **Reference Extraction via Custom AI Agent:** We developed a specialized Generative AI agent (a customized Gemini "GEM") specifically prompted to extract and structure the reference lists from the selected primary studies. This agent parsed the bibliographies to isolate paper titles, mitigating the formatting inconsistencies often found in PDF extractions.

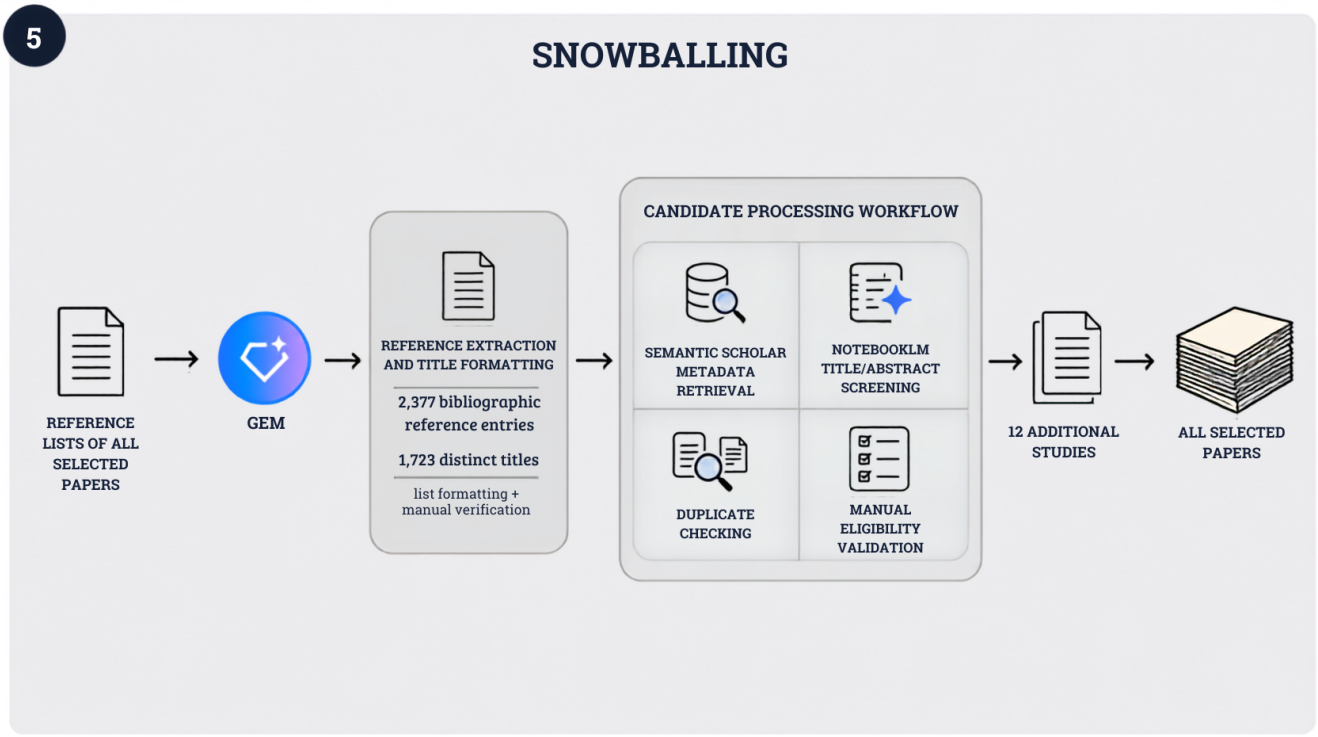


Figure 3. Starting from the reference lists of the 66 seed studies, 2,377 bibliographic reference entries were extracted, corresponding to 1,723 distinct titles. Candidate processing combined Semantic Scholar metadata retrieval, NotebookLM-assisted title/abstract screening, duplicate checking, and manual eligibility validation, resulting in 12 additional studies being included in the review.

2. **Automated Metadata Enrichment:** The extracted titles were processed through a custom Python script integrated with the Semantic Scholar API. For each title, the script performed a targeted search to retrieve the paper's metadata, including the official title, URL, and abstract. The output was a structured JSON dataset. To ensure data integrity, a manual verification step was performed by the first author of this work to resolve cases where the API could not find a match, with the first author manually appending missing entries.
3. **AI-Driven Screening of Candidate References:** The resulting JSON file, containing the titles and abstracts of all cited references, was uploaded to NotebookLM. We applied the same prompt used in the primary selection phase (see Figure 2), maintaining the core evaluation logic and inclusion criteria identical. The only modification was restricted to the output formatting instructions, transitioning from a semi-structured narrative format to a systematized tabular layout. This adjustment required the LLM to organize the evidence into discrete columns — Paper ID, Verdict (with justification), Tool/Method, Task/Agency, and Language — to streamline the processing of the large volume of data inherent to the snowballing phase without altering the underlying screening logic. The same criteria-oriented prompt was intentionally retained across both phases to preserve consistency in the underlying eligibility logic. In the snowballing stage, however, its output was interpreted as a preliminary relevance assessment based on the evidence available in titles and abstracts, rather than as a full-text eligibility judgment. Thus, the difference between the two phases concerned the depth of evidence available to support the assessment, not the inclusion and exclusion criteria themselves; final inclusion of snowballing candidates still required manual validation.

All technologies developed and used in this phase of the SLR are available in this study's replication package, thereby enabling future researchers to reuse them.

While the primary AI-assisted selection phase (see Figure 1, Step 3) relied on full-text analysis using NotebookLM, the snowballing phase employed a title-and-abstract screening approach. This decision is methodologically justified by two factors:

- **Scalability and Feasibility:** Across the 66 seed studies, the reference-extraction stage yielded 2,377 bibliographic reference entries, corresponding to 1,723 distinct titles across the combined reference pool. Given this volume, full-text retrieval for every cited reference was considered impractical for the preliminary screening stage.
- **Standard SLR Practice:** As per established guidelines (Wohlin, 2014), the first pass of snowballing typically relies on abstracts to filter out obviously irrelevant work. By using NotebookLM to analyze these abstracts, we obtained a scalable preliminary filter that helped prioritize the most promising candidates for subsequent manual validation.

No eligibility-based pre-filtering was applied before this preliminary screening stage. References were therefore not removed beforehand based on publication year or the absence of specific LLM-related terms. Instead, a broader candidate pool was retained and subsequently assessed according to the predefined inclusion and exclusion criteria, rather

than by literal compliance with the database search string. The AI-assisted snowballing process identified several candidate papers. After duplicate control and final manual validation, 12 additional articles were included in the review. This expanded the corpus to 78 studies prior to the final full-text eligibility verification described in Section 3.5.1.

3.5 Data Extraction and Study Classification

At this stage of the review, data extraction was conducted through a structured scheme aimed at capturing both descriptive and analytical attributes of the studies under analysis. This process also supported the classification of studies into groups and subgroups used in the subsequent synthesis.

3.5.1 Final Study Assessment

Before initiating the data extraction phase, the set of included studies underwent a final eligibility verification during the full-text reading. This additional refinement ensured that all selected papers fully satisfied the inclusion and exclusion criteria when examined in detail beyond the screening stages.

During this verification step, 13 papers were excluded for specific methodological or scope-related reasons, as summarized in Table 3.

Five studies were removed under Exclusion Criterion EC2, as their primary methodology relied on fine-tuning or retraining model weights rather than employing generative LLMs as inference-time agents through prompt-based interaction. Two studies were excluded under Inclusion Criterion IC1 due to task misalignment, as their objectives did not directly address either code smell detection or refactoring execution.

Additionally, two studies were excluded under Inclusion Criterion IC2 because they did not present an original contribution involving the application, demonstration, evaluation, or substantive analysis of LLMs in the context of the target tasks.

Finally, four studies were identified as republications or extended versions of previously included work. In these cases, a version consolidation process was applied, retaining only the most comprehensive publication to avoid duplication and ensure that each research contribution was represented only once in the evidence synthesis.

3.5.2 Hierarchical Grouping and Sub-Categorization

To support a fine-grained and reproducible synthesis, the selected studies were organized using a hierarchical classification scheme (see Figure 4). At the top level, papers were grouped into three primary thematic categories based on their research focus:

- **Group 1 – Performance and Technical Studies:** Empirical studies evaluating the effectiveness, accuracy, or limitations of LLMs when applied to software engineering tasks related to code smells and refactoring.
- **Group 2 – Methodological and Educational Studies:** Works proposing conceptual frameworks, methodological guidelines, architectural patterns, or educational perspectives on the use of LLMs in code quality and refactoring contexts.
- **Group 3 – Human-Centric Studies:** Research examining developer interaction with LLM-based tools, including usability, trust, adoption, and qualitative user feedback.

Given their direct relevance to the research questions, studies in Group 1 were further classified using a two-dimensional scheme. First, they were categorized according to the system architecture employed:

- **Standalone LLM:** Approaches relying solely on the LLM, without explicit integration of external analysis tools or repositories.
- **Hybrid Systems:** Approaches combining LLMs with traditional static analysis tools, rule-based detectors, retrieval-augmented generation (RAG), or other auxiliary components.

Second, each architectural category was subdivided based on the functional objective of the LLM:

- **Detection Only:** Identification of code smells or refactoring opportunities.
- **Execution/Refactoring Only:** Generation or application of refactored code without explicit detection.
- **End-to-End (Detection and Execution):** Integrated pipelines that both identify issues and perform refactoring.

This hierarchical classification resulted in eight distinct data extraction groups:

- **Sub-groups 1–6:** Corresponding to the combination of two architectural types and three functional objectives within the technical domain (*i.e.,* two architectures × three tasks).
- **Sub-group 7:** Methodological and educational contributions (Group 2).
- **Sub-group 8:** Human-centric studies (Group 3).

Each study was assigned to exactly one group to ensure consistency and avoid overlap during synthesis.

3.5.3 Role in Systematic Synthesis

This multi-level classification strategy played a central role in structuring the systematic synthesis. For each group, dedicated extraction spreadsheets were created to capture standardized attributes such as task definition, LLM model used, prompting strategy, evaluation metrics, datasets, and reported limitations.

These spreadsheets functioned as condensed representations of the literature, enabling efficient cross-study comparison and rapid identification of works addressing specific topics (*e.g.,* detection of Feature Envy smell, use of retrieval-augmented generation, or automated refactoring pipelines). While the extracted tables constitute the empirical backbone of this review, the subsequent analysis extends beyond descriptive aggregation. The discussion and results, presented in the next section, integrate qualitative interpretations, cross-group comparisons, and thematic synthesis to provide a comprehensive and critical overview of the current research landscape.

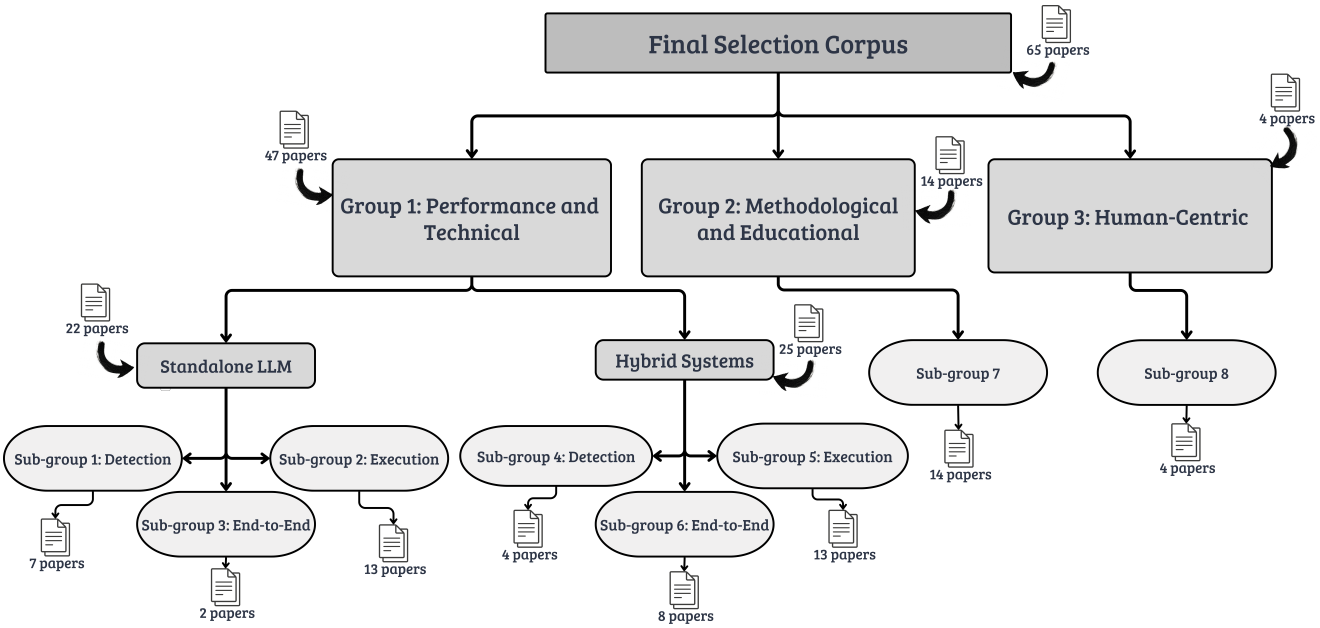


Figure 4. Hierarchical representation of the eight distinct data extraction groups.

4 Results and Discussion

This section presents the findings of the review in two complementary parts. First, a demographic and technical mapping summarizes the main characteristics of the included studies, such as model families, programming languages, publication venues, countries, and temporal distribution. Second, a critical synthesis examines recurring architectural patterns, evaluation practices, and reported limitations across the corpus. The discussion topics were derived directly from the research questions and from the analytical dimensions used during data extraction. Accordingly, the results do not address RQ1 and RQ2 as isolated standalone blocks, but through thematic syntheses that integrate technical, methodological, educational, and human-centered evidence across the selected studies. Unless otherwise stated, all findings reported in this section refer exclusively to studies on the prompt-based use of frozen-parameter LLMs included under the review protocol. Because the reviewed studies are heterogeneous in design, task scope, datasets, and metrics, the discussion emphasizes cross-study tendencies rather than direct meta-analytic comparison.

4.1 Demographic and Technical Mapping

This subsection summarizes the technical landscape and methodological preferences observed in the included studies. The goal is descriptive: to characterize the current literature before moving to the interpretive synthesis presented in the next subsection.

Which LLM models and families were most frequently employed in the studies? The analysis of the Large Language Model (LLM) families employed across the 65 included studies reveals a strong concentration around a small number of model providers, as illustrated by the frequency distribution in the research corpus (see Figure 5). One study in the corpus was classified as Not Reported for this specific variable because, although the paper clearly described the use of an LLM and therefore satisfied the inclusion criteria, it did not explicitly identify the model name or family adopted in the experiment.

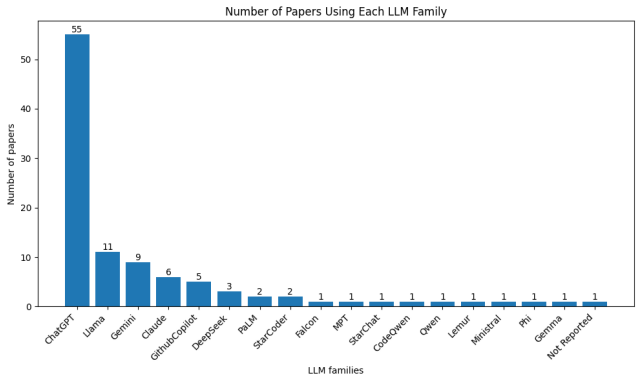


Figure 5. Number of papers in which each LLM family was used.

The ChatGPT family (OpenAI) was the most frequently reported model family, appearing in 55 of the 65 studies. This prevalence, corresponding to approximately 84.6% of the corpus, indicates that OpenAI models have occupied a central position in the current literature on code smell detection and refactoring. In particular, GPT-3.5 and GPT-4 were explicitly cited in 17 and 15 studies, respectively, making them the most commonly reported versions in empirical investigations of these tasks. One plausible explanation for this concentration is the early diffusion of OpenAI models through widely accessible APIs and their frequent use as experimental baselines in software engineering research, although the present review was not designed to determine the relative contribution of these factors.

The Llama family (Meta) appeared in 11 studies and was the most frequently reported open-source alternative in the corpus. The presence of Llama and its specialized derivatives, such as CodeLlama, suggests interest in settings where local deployment, greater control over execution, or experi-

mental reproducibility may be relevant considerations. However, the reviewed studies do not always make these motivations explicit, so this interpretation should be treated with caution.

In the intermediate range of usage, Gemini (Google) appeared in nine studies, followed by Claude (Anthropic) in six and GitHub Copilot in five. These models were more visible in recent publications and were often discussed in connection with larger context windows, coding assistance, or codebase-scale analysis capabilities. DeepSeek was identified in three studies, indicating a more recent but still limited presence in the reviewed literature.

Finally, the corpus also includes a long tail of less frequently used architectures, such as PaLM (n=2), StarCoder (n=2), Falcon (n=1), Phi (n=1), and Gemma (n=1). Although the reviewed literature remains concentrated around a few dominant proprietary and open-weight families, the presence of these additional models suggests ongoing experimentation with alternative architectures, including potentially lower-cost or more task-specific options.

Which programming languages and paradigms were primarily targeted in the studies? The analysis of the target programming languages reveals a significant concentration on a few dominant technologies, as illustrated in the distribution of languages across the research corpus (see Figure 6).

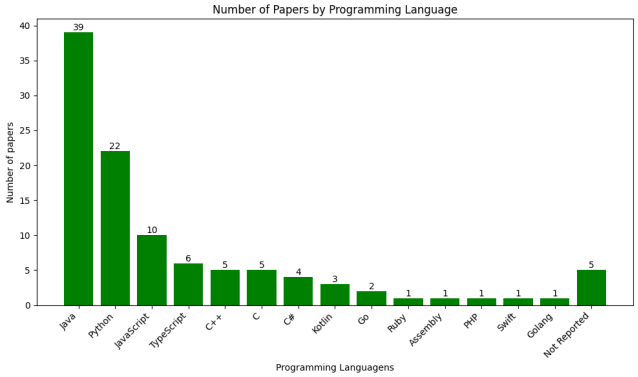


Figure 6. Number of papers in which each programming language was used.

Java emerged as the predominant language, appearing in 39 of the studies. This prominence likely reflects a combination of factors, including the availability of mature datasets and benchmarking tools commonly used in refactoring research, such as RefactoringMiner (Tsantalis et al., 2022) and Defects4J (Just et al., 2014), as well as the historical centrality of object-oriented systems in the code smell and refactoring literature. A further plausible explanation is that many classic refactoring definitions and examples, including those popularized by Fowler and Beck (1999), are closely associated with object-oriented structures that are strongly represented in Java-based software systems. However, the present review was not designed to isolate the relative weight of these factors empirically.

Python was the second most frequently reported language, appearing in 22 studies, followed by JavaScript (n=10), TypeScript (n=6), C++ (n=5) and C (n=5). Other languages, such as C# (n=4), Kotlin (n=3) and Go (n=2), also appear in the corpus, while a "long tail" of languages including Ruby, PHP, Swift, and Assembly appears in only one study each. Additionally, five studies were categorized as "Not Reported", as the specific programming languages targeted in their empirical evaluation were not explicitly identified in the text. Although these papers clearly addressed general-purpose programming contexts and satisfied the inclusion criteria, they did not specify which particular languages were used in their datasets, case studies, or experimental setups. For this reason, they were retained in the review but classified separately for this variable.

Overall, this distribution suggests a strong emphasis on object-oriented or object-centric programming contexts. Although multi-paradigm languages such as Python and JavaScript are increasingly represented, much of the reviewed literature still focuses on structures commonly associated with object-oriented design, such as classes, methods, and inter-class dependencies. This pattern indicates that LLM-based research on code smell detection and refactoring has so far concentrated more heavily on paradigms that align closely with the traditional refactoring literature.

At the same time, the corpus shows limited coverage of languages and paradigms that are not primarily organized around conventional object-oriented structures. Purely functional or function-first ecosystems were rarely represented, which suggests an opportunity for future research rather than a definitive absence of relevance. In other words, the current literature appears to have explored prompt-based LLM support more extensively in object-oriented settings than in paradigms such as functional programming, where notions of smell, modularity, and restructuring may follow different conceptual and practical assumptions.

Which specific code smells and refactoring techniques were the primary focus of the research? The analysis of research objectives reveals a strong concentration on the classic catalog of code smells and refactoring operations. Across the reviewed studies, the most frequently addressed targets were issues related to size, structure, and duplication, all of which occupy a well-established place in the software quality literature.

Among the identified code smells, *Code Clone* (19 papers) and *Long Method* (19 papers) were the most frequently reported, followed by *Large Class* (13 papers), which was often discussed under related labels such as *God Class* or *Blob*. As summarized in Table 4, these smells appear to dominate the literature reviewed in this study. One plausible explanation is that they are commonly associated with relatively explicit structural indicators, such as Lines of Code (LOC), duplication measures, or complexity-related signals, which may make them more amenable to operationalization in empirical studies. Although this interpretation cannot be established definitively from the corpus alone, it is consistent with the broader historical prominence of these smells in software engineering research.

This pattern also aligns, at least in part, with prior review evidence. Sobrinho et al. (2021), in a systematic review of code smell research published between 1990 and 2017, also identified strong attention to traditional smell categories. The comparison is analytically useful because it suggests continuity in the underlying research focus, even though the more recent literature reviewed here introduces LLM-based meth-

ods that were absent from earlier generations of studies.

In addition to these more localized or structurally explicit smells, *Feature Envy* appeared in nine papers and occupies a somewhat different position in the corpus. Unlike clone-related or size-related smells, Feature Envy typically requires reasoning about inter-class dependencies, distribution of behavior, and coupling relationships. Its presence in the reviewed studies is therefore noteworthy because it points to a shift from predominantly local detection and transformation tasks toward scenarios that demand broader structural reasoning.

In terms of refactoring techniques, the literature reveals a direct correspondence with the targeted code smells. The most frequently reported operations include *Extract* (29 papers), *Renaming* (17 papers), and *Move* (10 papers), followed by Inline operations (seven papers). These findings are broadly consistent with the refactoring literature surveyed by Abid et al. (2020), in which the authors reviewed 3,183 papers on refactoring published up to May 2020. In the present corpus, these operations appear not only as common implementation mechanisms, but also as direct responses to the smells most frequently studied. For example, *Extract Method* is closely associated with *Long Method*, whereas *Move Method* is a standard intervention for *Feature Envy-like* coupling problems.

Beyond these classic operations, the corpus also shows signs of emerging interest in language-idiomatic and modernization-oriented transformations. Examples include the adoption of Pythonic idioms (Midolo and Di Penta, 2025) [P219] and the replacement of traditional Java loops with Stream-based constructs (Midolo and Tramontana, 2025) [P131]. These cases remain less frequent than classic smell-driven refactorings, but they are analytically relevant because they suggest that some recent studies are extending LLM-based refactoring beyond structural cleanup toward idiomatic improvement and language modernization. Even so, this trend should be interpreted as emergent rather than dominant in the current literature.

Overall, the concentration on a relatively small set of classic smells and corresponding refactoring operations suggests that the current state of the art is still centered on evaluating LLMs against well-established software quality problems. This focus is understandable from an empirical standpoint, since canonical smells and refactorings offer clearer benchmarks, more stable terminology, and more interpretable evaluation settings than broader architectural transformations. At the same time, the limited presence of more system-wide or paradigm-specific transformations indicates that the literature has not yet moved far beyond the traditional core of smell-driven maintenance tasks.

Although Table 4 presents only the four most prevalent code smells and refactoring strategies identified in the papers analyzed in this SLR, the replication package of this study provides access to the complete list of smells and refactorings, as well as traceability information indicating the source papers from which they were extracted.

How are studies on LLM-based code smell detection and refactoring distributed across publication venues? The selected primary studies are distributed across 46 distinct publication venues and publication outlets, ranging from major software engineering conferences to specialized workshops, journals, preprints, and a small number of book chapters. Table 5 summarizes the venues with the highest frequency. This dispersion suggests that research on LLM-based code smell detection and refactoring is not concentrated in a single publication channel, but is instead being discussed across multiple segments of the software engineering community.

Table 4. Most Prevalent Code Smells and Refactoring Techniques in the Primary Studies.

Code Smells	#	%	Refactoring Techniques	#	%
Code Clone	19	29.2%	Extract (Method/Class)	29	44.6%
Long Method	19	29.2%	Renaming	17	26.2%
Large Class	13	20.0%	Move (Method/Class)	10	15.4%
Feature Envy	9	13.8%	Inline (Method/Var)	7	10.8%

Among the identified venues, the International Conference on Software Engineering (ICSE) accounts for the largest share, with seven papers (10.8% of the corpus). This prominence indicates that the topic has gained visibility in one of the most established and competitive venues in software engineering. At the same time, the absolute number remains modest relative to the overall corpus, which suggests growing interest rather than mature dominance within a single flagship outlet.

A second notable pattern is the presence of studies in venues traditionally associated with software maintenance, evolution, and reengineering, particularly the International Conference on Software Analysis, Evolution and Reengineering (SANER), with three publications, and the International Conference on Software Maintenance and Evolution (ICSME), with two publications. This distribution is analytically meaningful because it places LLM-based refactoring research within the broader tradition of software maintenance studies, rather than treating it as an isolated AI application domain. In other words, the corpus suggests that research on LLM-based code smell detection and refactoring is emerging not outside software engineering, but from within its established maintenance-oriented concerns.

The corpus also includes five studies disseminated through arXiv, all of which were identified exclusively through the backward-snowballing process rather than through the initial database search. Their presence is noteworthy in light of the rapid pace of publication in the LLM ecosystem, where preprints may provide early access to emerging empirical results. Although arXiv submissions are not necessarily peer-reviewed, publication status was not used as an absolute eligibility criterion in this review, and these studies were retained because they satisfied the predefined substantive inclusion criteria. Notably, two of the five preprints were subsequently accepted for publication at peer-reviewed venues: P388 (Dilhara et al., 2024) and P389 (Cordeiro et al., 2026). This provides additional context regarding the subsequent publication trajectory of part of this subset, although the review status of the evidence remains non-uniform across all five studies. Therefore, findings derived from preprints should still be interpreted with appropriate caution.

In addition, the review identified four book chapters retrieved through the same database-driven selection process described in the methodology. Although they do not represent a dominant publication format in the corpus, their pres-

Table 5. Distribution of Primary Studies by Publication Venue.

Publication Venue	Acronym	Type	#	%
International Conference on Software Engineering	ICSE	Conference	7	10.8%
arXiv (Preprints)	-	Repository	5	7.7%
International Conference on Software Analysis, Evolution and Reengineering	SANER	Conference	3	4.6%
International Workshop on Integrated Development Environments	IDE	Workshop	3	4.6%
International Conference on Software Maintenance and Evolution	ICSME	Conference	2	3.1%
International Conference on Program Comprehension	ICPC	Conference	2	3.1%
International Conference on Mining Software Repositories	MSR	Conference	2	3.1%
International Conference on Automated Software Engineering	ASE	Conference	2	3.1%
International Conference on AI Engineering	CAIN	Conference	2	3.1%
Others (37 venues with 1 paper each)	-	Mixed	37	56.9%
Total	-	-	**65**	**100%**

ence indicates that relevant discussions on LLM-based refactoring are also appearing in editorial collections and broader scholarly compilations, rather than exclusively in conference and journal venues.

Finally, the presence of publications in more specialized venues, such as the IEEE/ACM International IDE Workshop and the Conference on AI Engineering (CAIN), suggests a complementary trend toward tooling integration and engineering-oriented applications of LLMs. These venues appear to capture studies that are closer to developer workflows, IDE integration, and practical deployment scenarios. Taken together, the venue distribution suggests that LLM-based refactoring is currently positioned at the intersection of mainstream software engineering, maintenance research, and emerging AI-engineering-oriented publication spaces.

What is the geographical distribution of the papers included in this SLR? To analyze the global distribution of research contributions, the primary affiliation of the first author was used as a proxy for determining the country of origin of each study. As presented in Table 6, the results reveal a marked concentration of studies in North America, which accounts for 40.0% of the total corpus when aggregating contributions from the *United States* (23.1%) and *Canada* (16.9%). This pattern suggests that a substantial share of the reviewed literature on LLM-based code smell detection and refactoring has been produced in that region.

Table 6. Geographical Distribution of Research (by First Author Affiliation).

Country	#	%
USA	15	23.1%
Canada	11	16.9%
China	8	12.3%
Brazil	5	7.7%
Italy	4	6.2%
Independent / Non-affiliated	3	4.6%
Sweden	2	3.1%
Serbia	2	3.1%
Others (15 countries with 1 paper each)	15	23.1%
Total	**65**	**100%**

China appears as the most represented country in Asia, accounting for 12.3% of the selected studies. *Brazil* occupies a notable position in the corpus, ranking fourth overall with five publications (7.7%). This indicates that research on LLM-based code smell detection and refactoring is not restricted to the main North American and East Asian centers, but also includes visible contributions from other regions.

In addition, three studies (4.6%) were authored by researchers identified as independent or without a clearly stated institutional affiliation. Although this category should be interpreted cautiously, its presence suggests that some contributions in this domain may be emerging outside conventional academic affiliation structures.

The remaining 23.1% of the studies are distributed across 15 different countries, indicating that, despite visible concentration in a small number of locations, the literature exhibits a broader international spread. Overall, the geographical distribution suggests that research on LLM-based code smell detection and refactoring is internationally distributed, although unevenly represented across regions.

How has research on LLM-based code smell detection and refactoring evolved over time? To examine the temporal evolution of research activity in this domain, we analyzed the distribution of the selected studies by publication year. As shown in Table 7, the corpus exhibits a marked increase over time, particularly from 2023 onward.

Table 7. Temporal Distribution of Primary Studies.

Year	#	%
2026	1	1.5%
2025	34	52.3%
2024	23	35.4%
2023	7	10.8%
Total	**65**	**100%**

Only seven studies were published in 2023, followed by a substantial increase to 23 in 2024 and 34 in 2025. The latter alone accounts for 52.3% of the total corpus. This distribution indicates that research on LLM-based code smell detection and refactoring has expanded rapidly in a short period, with the strongest concentration occurring in the most recent year of the review window.

The increase observed between 2023 and 2025 suggests that the field is still in an active growth phase. One plausible explanation is the broader availability of large language models and their growing incorporation into software engineering workflows during this period. However, this interpreta-

tion should be treated cautiously, since the present review describes the temporal distribution of the literature rather than directly measuring the causes of publication growth.

The corpus also includes a single study with a 2026 publication date, namely P080 (Melo et al., 2026). Although its final bibliographic assignment corresponds to the January–February 2026 issue of IEEE Software, IEEE Xplore records October 17, 2025, as its date of publication. Therefore, P080 was already publicly available and retrievable when the automated search was conducted on November 5, 2025, and its representation under 2026 reflects its final issue year.

Overall, the temporal distribution reinforces the interpretation that LLM-based code smell detection and refactoring is a recent and rapidly expanding line of research, with most of the current evidence emerging only in the last two years of the analyzed period.

4.2 Critical Synthesis of Findings

This subsection moves beyond the descriptive mapping presented earlier and synthesizes the reviewed evidence from a qualitative and analytical perspective. Its purpose is to examine the main patterns observed across the corpus, with particular attention to technical implications, architectural choices, evaluation practices, and recurring limitations reported in the literature.

Rather than treating the included studies as directly comparable in all respects, this synthesis focuses on identifying recurring tendencies, points of convergence, and important differences across the reviewed evidence. In this sense, the discussion aims not only to summarize what has been studied, but also to clarify how different research approaches have framed and evaluated the use of LLMs in code smell detection and refactoring.

To support a more organized interpretation of the literature, the synthesis follows the three classification groups established during data extraction: **(i) Technical Performance and Benchmarking (Group 1)**; **(ii) Methodological and Educational Frameworks (Group 2)**; and **(iii) Human Factors and Developer Interaction (Group 3)** (see Figure 4). This structure is intended to facilitate the discussion of different forms of evidence without conflating experimental performance, methodological guidance, and socio-technical concerns. It also helps make visible the research gaps that emerge at each level, from model-centered evaluation to the broader conditions required for effective human-in-the-loop use.

4.2.1 Group 1: Technical Performance and Benchmarking

This group synthesizes the studies that evaluate LLM-based approaches primarily in terms of technical performance, including detection quality, refactoring correctness, and architectural robustness. Because this is the most quantitatively developed portion of the corpus, the discussion emphasizes recurring technical patterns rather than isolated results.

How do standalone and hybrid LLM-based approaches compare in detecting and addressing structurally complex code smells such as Feature Envy? A recurring pattern in the reviewed literature is that structurally complex smells, especially those involving inter-class dependencies, remain difficult to handle through standalone LLM prompting. This limitation is particularly visible in studies addressing *Feature Envy*, where reliable detection and refactoring depend not only on local semantic interpretation, but also on the ability to reason about cohesion, coupling, and method placement across class boundaries. In this sense, the reviewed evidence suggests that semantic inference by itself is often insufficient for high-confidence performance in these scenarios.

The empirical results reported in the corpus reinforce this interpretation. In studies evaluating standalone approaches, performance on *Feature Envy* detection remained limited, with F1-scores ranging from 0.035 to 0.19 in Mesbah et al. (2025) [P242], and reaching 0.29 in the best critical-severity setting reported by Silva et al. (2024) [P101]. Although these results come from different experimental settings and should not be directly aggregated, they point in the same direction: prompt-only or standalone LLM approaches appear to struggle when the task requires broader structural reasoning beyond the immediately provided code fragment.

By contrast, the reviewed studies suggest that hybrid architectures tend to perform more reliably in this class of problem. In particular, approaches that combine LLMs with auxiliary structural models or analysis layers—such as Dual-LSTMs, Graph Neural Networks, dependency analysis, or IDE-based verification—were associated with better results in the reviewed corpus. Akalanka et al. (2025) [P213], for example, report an accuracy of 71% and an F1-score of 0.70 in a hybrid setting. Although cross-study heterogeneity prevents stronger quantitative claims, the contrast with weaker standalone results suggests that structural support mechanisms may substantially improve performance in Feature Envy-related tasks.

A plausible explanation for this difference is that hybrid systems reduce the burden placed on the LLM to infer architectural relationships from raw code alone. Instead of relying exclusively on prompt-based semantic interpretation, these approaches introduce intermediate representations or filtering stages that expose structural properties more explicitly, such as AST-derived coupling, dependency graphs, or feasibility checks. In practice, this means that the LLM is guided by a more constrained and diagnostically meaningful input, which may reduce the generation of superficially plausible but structurally invalid recommendations.

This issue is closely related to hallucination and operational invalidity in refactoring suggestions. Bellur et al. (2025) [P023] report that, in both synthetic and real-world datasets, only 20% of LLM-generated *Move Method* recommendations were valid, while the remaining 80% were considered hallucinated or infeasible. This result is particularly important because *Move Method* is one of the canonical refactorings used to address *Feature Envy*. In this context, the study provides strong evidence that raw LLM recommendations cannot be treated as reliable refactoring actions without additional structural filtering. The hybrid strategy adopted in MM-ASSIST mitigates this problem by using IDE-native static analysis and feasibility checks before presenting a suggestion to the developer.

Taken together, the reviewed studies suggest that the main

limitation of standalone LLM use in *Feature Envy-related* scenarios is not merely low predictive accuracy, but insufficient access to project-level structural context. Hybrid approaches appear to address this limitation by combining semantic generation with structural verification or learned structural representations. Therefore, while LLMs may contribute useful semantic intuition for identifying refactoring opportunities, the literature reviewed here indicates that reliable handling of structurally complex smells depends, in many cases, on some form of architectural or tool-supported augmentation rather than on standalone prompting alone.

Finding #1: Standalone vs. Hybrid Approaches

Standalone prompting tends to struggle with structurally complex, inter-class smells such as *Feature Envy*. Across the reviewed studies, hybrid approaches that combine LLMs with structural analysis, auxiliary models, or IDE-based verification appeared more reliable because they expose dependencies and feasibility constraints that are not available in local prompts alone.

How do RAG-enhanced and non-grounded architectures compare in LLM-based code smell detection and refactoring? Across the reviewed studies, repository-aware or retrieval-augmented workflows were often associated with better outcomes than non-grounded prompting in code smell detection and refactoring tasks. This pattern was particularly visible in studies involving structurally complex transformations, where access to project-level context appeared to improve compilation success, test-passing performance, and task-specific accuracy. However, these findings should be interpreted cautiously, since the reviewed studies differ substantially in task design, datasets, target smells, programming languages, and evaluation metrics.

One recurring pattern in the corpus is that non-grounded prompting becomes less reliable when the task requires information that extends beyond the immediately provided code fragment. In such cases, models may produce locally plausible outputs that fail to preserve broader structural dependencies. By contrast, RAG-enhanced approaches attempt to reduce this limitation by injecting repository-specific context, structural descriptions, or previously retrieved examples into the prompt. In comparative settings, this strategy was often associated with improved performance. For example, Xu (2025) [P032] reports higher test-passing rates for a RAG-augmented workflow than for a raw prompting baseline, with results improving from 23/37 to 31/37 in the evaluated refactoring tasks.

A similar pattern appears in smell-oriented settings. Wu et al. (2024) [P003], for instance, report substantial gains when project-specific metadata and contextual signals are incorporated into the generation process, including a marked improvement in F1-score for smell detection relative to less grounded configurations. Likewise, Guo et al. (2025) [P061] suggest that retrieval-supported prompting can outperform simpler baselines, including non-grounded and randomly sampled few-shot strategies, in intention-oriented refactoring scenarios. Taken together, these studies suggest that grounding the model in repository-relevant context often improves reliability relative to prompt-only baselines, especially when the task depends on structural consistency or project-specific conventions.

That said, the evidence should not be interpreted as proving a uniform advantage of RAG in every setting. The reviewed studies vary too widely to support pooled quantitative claims, and some improvements may reflect task-specific design choices rather than a generalizable effect size. The most defensible conclusion is therefore that repository-aware grounding emerges in the corpus as a recurrent and promising strategy for improving the performance of LLM-based smell detection and refactoring, particularly in contexts where direct prompting lacks sufficient structural context.

Finding #2: RAG vs. Non-Grounded Architectures

Repository-aware grounding was frequently associated with better outcomes than non-grounded prompting in code smell detection and refactoring, especially when tasks depended on project-level context. The reviewed evidence suggests improvements in structural validity, test passing, and task-specific accuracy, although cross-study heterogeneity limits stronger pooled claims.

Through which mechanisms do RAG-enhanced pipelines appear to improve reliability in LLM-based code smell detection and refactoring? Beyond quantitative comparisons, the reviewed studies suggest that retrieval-augmented workflows improve reliability through a small set of recurring mechanisms. Rather than treating retrieval only as a way of adding more text to the prompt, the literature frames RAG as a means of supplying the model with more relevant, more constrained, and more task-specific context than would otherwise be available through direct prompting alone.

A first mechanism is the use of retrieved examples or templates as contextualized guidance. In this configuration, the model is not expected to infer a transformation pattern solely from general pretraining knowledge. Instead, it is supported by examples, refactoring instances, or project-relevant patterns retrieved on the basis of semantic or structural similarity. Studies such as Guo et al. (2025) [P061] and Xu (2025) [P032] suggest that this strategy can improve alignment with the intended task and reduce the variability typically associated with unconstrained generation.

A second mechanism is the injection of domain-specific knowledge into the generation process. Some studies retrieve external guidance, such as refactoring definitions, tool documentation, or design-oriented references, in order to constrain the model's reasoning and terminology. Pandini et al. (2025) [P225], for example, indicate that retrieval-supported workflows can provide a clearer rationale for individual transformations by grounding the model in explicit refactoring knowledge rather than relying exclusively on internal model priors. In this sense, retrieval may help transform the interaction from open-ended generation into a more guided maintenance process.

A third mechanism concerns the handling of complex structural dependencies. Tasks such as *Move Method*, *God Class* decomposition, or broader smell resolution often require access to relationships that exceed a narrow prompt

window, including call chains, cross-class dependencies, or responsibility distribution across modules. In these cases, RAG-based systems attempt to expose the surrounding structural context more explicitly through indexed code elements, dependency descriptions, or repository-level metadata. This appears particularly relevant in studies such as Wu et al. (2024) [P003], where retrieval-supported refactoring was associated not only with improved smell detection performance, but also with reductions in CK metrics after the transformation of *God Classes*.

Many studies suggest that retrieval-enhanced outputs are more structured, justifiable, and maintainable than those generated without contextual grounding. Rather than simply producing candidate transformations, RAG-enhanced systems better support identifying why a change is appropriate and how it relates to existing structural patterns in the codebase. Although this evidence remains heterogeneous and should not be overstated, it indicates that the benefits of retrieval may extend beyond accuracy to include output interpretability and maintainability.

Overall, the reviewed literature suggests that the contribution of RAG is not merely to increase the amount of context available to the model, but to improve the relevance and usability of that context for smell detection and refactoring tasks. The most persuasive evidence in the corpus, therefore, supports a view of RAG as an enabling mechanism for context-sensitive, structurally informed, and more reliable LLM-assisted maintenance workflows, rather than as a universally superior solution in all settings.

Finding #3: Why Does RAG Help?

RAG improves reliability by providing more relevant context, such as task-specific examples, repository-level structure, and domain knowledge, which helps constrain generation when smells and refactorings exceed the local prompt window.

4.2.2 Group 2: Methodological Frameworks and Educational Applications

This group covers studies that frame LLM use as a workflow, methodological strategy, or learning support mechanism rather than only as a technical artifact. The evidence here is more heterogeneous and exploratory, so the synthesis focuses on how prompting, validation, automation design, and educational interaction are structured in practice.

What automation pipeline structures and validation mechanisms have been proposed in the literature to improve the reliability of prompt-based LLM code smell detection and refactoring? The reviewed studies suggest that reliable LLM-driven code smell detection and refactoring rarely result from direct prompt-response interaction alone. Instead, a recurring pattern in the literature is the transition from snippet-level assistance toward workflow-centric automation frameworks that combine task decomposition, external verification, and, in more advanced cases, feedback-driven regeneration. Taken together, these studies indicate that raw LLM output is often too unstable to support high-confidence smell-related analysis and refactoring without additional control mechanisms.

A first recurring design principle is the orchestration of code smell analysis and refactoring as a structured pipeline rather than as a single generation step. In this approach, complex maintenance tasks are decomposed into sequential stages, such as smell identification, candidate transformation, validation, and final integration. Sonkin and Tudose (2025) exemplify this direction through the Prompt Pipeline Language (PPL), a declarative format for defining chained prompt workflows in which the output of one stage becomes the input to the next. According to the authors, this form of structured execution helps isolate errors early and reduces the risk that flawed intermediate outputs propagate downstream. More broadly, this evidence suggests that workflow decomposition is not merely an implementation choice, but a reliability strategy for constraining LLM behavior in smell-oriented and refactoring-oriented scenarios.

A second recurring element is the use of external validation mechanisms to close the automation loop. Across the reviewed studies, raw LLM suggestions were frequently treated as provisional outputs requiring further verification before acceptance. Cordeiro et al. (2024) [P389], for example, emphasize the importance of automated test generation when existing oracles are insufficient to assess behavior preservation. Their discussion highlights that, without adequate unit tests or equivalent validation artifacts, determining whether an LLM-generated refactoring preserves functionality remains difficult and often depends on manual inspection. In this sense, the proposed integration with tools such as *EvoSuite* or *Randoop* reflects a broader pattern in the literature: reliability increases when LLM-generated smell-related recommendations and refactoring actions are coupled with explicit behavioral checks rather than assumed from the model output itself.

The same logic appears in studies that integrate LLM-based smell analysis and refactoring support with static analysis and IDE-native verification. Bellur and Batole (2025) report that many initial suggestions produced by the foundation model were invalid in tasks such as *Move Method* recommendation. To mitigate this problem, their approach uses IDE APIs and static analysis capabilities to verify whether a suggested refactoring is structurally feasible before presenting it to the developer. In practice, the model is not allowed to operate as an unconstrained generator; instead, its suggestions are filtered through compile-time and type-accessibility checks. This design is important because it illustrates a broader architectural lesson from the corpus: in complex smell-resolution and refactoring scenarios, reliability depends not only on generation quality, but also on whether the surrounding toolchain can reject hallucinated or non-executable transformations before they reach the user.

A more advanced variation of this pattern is the introduction of feedback-driven or self-healing pipelines. Sonkin and Tudose (2025) describe a workflow in which failed validation triggers rollback, error logging, and a new generation attempt conditioned on the detected failure. Rather than requiring immediate manual debugging, the system reuses validation feedback as input for iterative correction until the issue is resolved or escalated to human intervention. In the re-

ported setting, this closed-loop strategy was associated with improvements in implementation correctness and reductions in compilation errors. Although this evidence comes from a specific framework rather than from the corpus as a whole, it is analytically important because it shows how reliability can be increased when generation is embedded in a controlled cycle of execution, diagnosis, and repair.

Overall, the reviewed studies point to a common conclusion: the reliability of LLM-driven code smell detection and refactoring depends less on isolated prompt quality than on pipeline design. More specifically, the most promising approaches are those that treat smell analysis and refactoring as validated workflows, combining decomposition, behavioral or structural verification, and, where possible, iterative recovery from failure. This does not mean that fully autonomous smell detection and refactoring are already stable realities across the literature. Rather, the evidence suggests that higher reliability is achieved when LLMs are embedded in orchestration layers that constrain, check, and, when necessary, correct their outputs before integration.

Finding #4: Reliable Smell Detection and Refactoring Pipelines

Reliable LLM-driven code smell detection and refactoring are typically implemented as validated workflows rather than as single prompt-response steps. The strongest approaches combined task decomposition, static analysis, testing, and feedback-driven regeneration to filter hallucinations and recover from invalid recommendations or transformations before they reached the developer.

How are educational studies using LLMs to support the learning of code quality and refactoring, and what interaction patterns emerge when students use prompt-based LLMs in these tasks? The reviewed educational studies suggest that LLMs are being used less as autonomous teaching systems and more as interactive supports for learning activities related to code quality analysis and refactoring. In these settings, the educational value of the model appears to depend not only on what the LLM can generate, but also on how learners structure the interaction and interpret the resulting outputs. Taken together, the reviewed evidence indicates that educational uses of LLMs in this domain are closely tied to prompting behavior, reflective evaluation, and the learner's active role in the problem-solving process.

Menolli et al. (2024) [P109], for example, frame LLM-assisted refactoring through the lenses of single-loop and double-loop learning. In their study, students were encouraged not only to prompt the model to produce refactoring suggestions, but also to analyze, question, and refine the generated outputs. The reported results suggest that this interaction can support aspects of computational thinking, particularly problem formulation, decomposition, and abstraction. Rather than simply delegating the task to the model, students needed to structure the problem in a way that made prompting effective and then critically assess whether the proposed transformation was appropriate.

A complementary perspective is offered by Oliveira et al. (2025) [P112], who identify different patterns of student interaction with LLMs during refactoring activities. Their findings suggest that learning outcomes vary depending on how students position the model within their workflow. In the so-called starter pattern, students first attempt the task independently and then consult the LLM for targeted support; this strategy was associated with better refactoring outcomes in the reported setting. By contrast, the navigator pattern—where the student relies on the LLM as the main driver of the task through step-by-step prompting—was associated with weaker results. This distinction is analytically important because it suggests that LLMs may be more educationally valuable when used as reflective assistants than when treated as primary solution generators.

Finding #5: Educational Use of LLMs

Educational studies suggest that LLMs are most useful for code quality and refactoring learning when students remain active problem solvers. Reflective use, targeted prompting, and iterative evaluation appear to support learning, whereas passive delegation to the model tends to reduce both solution quality and conceptual understanding.

Miño et al. (2024) [P311] reinforce this interpretation from a related instructional context involving LLM-assisted analysis in security-aware coding education. Their study suggests that models such as ChatGPT and Bard may help identify more common problems more quickly than students, but may still struggle with more complex structural or architectural issues unless the prompting is sufficiently precise. Although this study is somewhat broader than refactoring alone, it remains relevant to the present synthesis because it highlights a recurring educational limitation: the usefulness of the model depends heavily on the learner's ability to formulate targeted prompts and to critically inspect what the model returns.

Overall, the reviewed educational evidence suggests that the pedagogical value of LLM-based support in code quality and refactoring tasks depends strongly on interaction design and learner behavior. When students remain actively engaged in decomposition, evaluation, and revision, LLMs appear to support reflective learning and problem-solving practice. Conversely, more passive reliance on model-generated solutions may reduce the quality of the outcome and risk reinforcing misconceptions or shallow understanding. In this sense, the literature reviewed here points toward an educational use of LLMs that is most effective when the model is embedded in a guided, reflective, and human-led learning process.

4.2.3 Group 3: Human Factors, Interaction, and Developer Perception

This group brings together studies concerned with developer behavior, trust, interaction patterns, and adoption in LLM-assisted code quality and refactoring tasks. Although smaller than the technically oriented group, this body of evidence is important because it highlights human-AI collaboration as a central condition for practical use.

How do developers interact with LLM-based refactoring tools, and what design implications emerge for trustworthy human-AI collaboration? The reviewed empirical studies suggest that the use of LLMs in refactoring-related workflows is characterized by a tension between high perceived usefulness and limited trust in fully autonomous code generation. In other words, developers often report interest in using these tools as part of their maintenance activities, but remain cautious about accepting generated changes without review. This pattern is important because it implies that the practical value of LLM-based refactoring tools may depend less on full automation than on how effectively they support iterative, review-centered collaboration.

A first recurring theme in the literature concerns interaction patterns and user intentionality. Studies such as Alomar et al. (2024) [P011] and Deo et al. (2024) [P013] indicate that developers frequently alternate between very general requests and more granular, instruction-driven prompting. Vague requests such as "refactor this code" may rely on the model to infer the intended quality objective, but this low-context interaction often produces outputs that require subsequent clarification or revision. As a result, refactoring with LLMs is often not a single-step act, but an iterative conversational process. This is consistent with the evidence reported by Deo et al. (2024) [P013], who observed that pull request discussions involving ChatGPT required, on average, 4.6 prompts before reaching a conclusion. Taken together, these findings suggest that developers tend to use LLMs not as one-shot refactoring engines, but as participants in an extended dialogue of proposal, feedback, and adjustment.

A second theme concerns trust and the limited direct adoption of generated code. The reviewed evidence suggests that developers are willing to engage with LLM outputs, but often prefer to adapt, refine, or reinterpret them rather than accept them unchanged. Das et al. (2025) [P026], for example, report that only a small proportion of AI-generated code for issue resolution was incorporated without modification, whereas a much larger share of successful interactions focused on refining existing code. Although this result comes from a broader maintenance setting and should not be generalized mechanically to all refactoring scenarios, it remains analytically relevant because it supports a recurring pattern in the corpus: LLMs appear to be used more frequently as tools for ideation, revision, and assistance than as autonomous producers of final changes.

This cautious stance is reinforced by concerns about behavioral correctness and unintended side effects. Borg (2025) [P039], for instance, highlights the risk that AI-generated refactorings may introduce unintended behavioral changes or new vulnerabilities if accepted uncritically. From a human-factors perspective, this is significant because it suggests that trust in LLM-based refactoring should not be understood as simple acceptance or rejection. Rather, the literature points to the importance of calibrated trust: developers must neither rely on the model blindly nor dismiss it prematurely, but instead engage with its outputs through structured validation and review.

These observations carry direct implications for tool design. The reviewed studies suggest that effective LLM-based refactoring tools should be designed to support iterative interaction, make the rationale for suggested changes more explicit, and facilitate human validation rather than attempting to eliminate it. In this sense, the literature points toward design strategies centered on transparency, revision support, and controllable interaction. Rather than optimizing only for autonomous generation, future tools may be more effective when they are designed as collaborative environments in which developers can query, inspect, refine, and validate model suggestions in context.

Overall, the reviewed evidence suggests that the most promising role for LLMs in refactoring practice is not that of an autonomous replacement for developer judgment, but that of a conversational and revisable assistant embedded in a human-in-the-loop workflow. This interpretation is consistent with the broader pattern observed across the corpus: practical adoption appears to increase when the model supports developer reasoning and iteration, rather than when it is expected to produce trustworthy final code in a single step.

Finding #6: Human-AI Collaboration

Developers often perceive LLMs as useful refactoring assistants, but trust in fully autonomous code generation remains limited. The literature therefore points toward tools that support iterative dialogue, transparent rationale, validation, and visible human control rather than one-shot automation.

Taken together, these findings provide explicit answers to the two research questions. **Regarding RQ1, prompt-based frozen-parameter LLMs have been used for code smell detection both as standalone semantic analyzers and, increasingly, as components of hybrid or repository-aware workflows.** The evidence synthesized in *Findings #1–#3* shows that standalone prompting can identify smell-related patterns but remains less reliable when detection depends on structural and cross-class information, as illustrated particularly by Feature Envy. Hybrid architectures and retrieval-augmented approaches address part of this limitation by exposing structural dependencies, repository context, examples, and domain knowledge to the model. *Finding #4* further shows that detection becomes more dependable when embedded in validated workflows that combine LLM reasoning with static analysis, decomposition, and external verification. Across the reviewed studies, these approaches were evaluated using metrics such as precision, recall, F1-score, and accuracy, although heterogeneity in tasks and datasets limits direct quantitative comparison. The broader evidence from *Findings #5* and *#6* also indicates that effective use depends on maintaining active human reasoning, interpretable interaction, and appropriate oversight rather than relying on autonomous model judgments.

Regarding RQ2, prompt-based frozen-parameter LLMs have been used for refactoring to recommend, generate, and support the execution of structural code transformations, ranging from individual operations to integrated detection-and-refactoring workflows. *Findings #1–#3* indicate that refactoring reliability improves when the model has access to structural or repository-level context, particularly for transformations involving depen-

dencies beyond the local code fragment. *Finding #4* shows that the most reliable approaches treat generated refactorings as provisional outputs and subject them to compilation, testing, static analysis, feasibility checks, or feedback-driven regeneration before acceptance. Beyond technical correctness, *Finding #5* indicates that LLM-supported refactoring can also serve as a learning activity when users critically evaluate and revise generated suggestions, while *Finding #6* shows that developers tend to favor iterative and revisable assistance over fully autonomous execution. Thus, across both research questions, the reviewed evidence suggests that the most dependable use of prompt-based LLMs is not as isolated generators or detectors, but as context-aware components of validated, iterative, and human-supervised maintenance workflows.

5 Implications: Guidelines for Tools

Based on the patterns identified in this SLR, we argue that future tools for LLM-based code smell detection and refactoring should be designed not as generic chat interfaces, but as context-aware, workflow-oriented, and human-supervised development assistants. The reviewed literature suggests that tool effectiveness depends less on raw model capability alone than on how the LLM is embedded into the developer's environment, how its outputs are constrained and validated, and how much control remains with the human user.

A first implication concerns the **practical attractiveness of prompt-based integration itself**. Because this review focused on prompt-based use of LLMs rather than fine-tuned or retrained models, we do not claim that prompt-based tools are universally superior to adapted-model approaches. However, within the scope of the reviewed evidence, prompt-based integration offers important engineering advantages for tool builders. In particular, tools can be integrated through provider APIs without requiring model retraining pipelines, making it easier to adopt newer models, switch providers, and support user-managed access models such as *bring your own key*. This model-agnostic architecture may lower adoption barriers and accelerate experimentation in real development settings, especially when the goal is to augment developer workflows rather than build domain-specialized models from scratch.

A second implication is that **deep integration with the developer's environment is essential**. One of the clearest findings of the review is that LLM performance degrades when the model receives only fragmentary or manually assembled context. For this reason, future tools should be connected directly to the IDE or repository environment so that they can automatically extract the relevant class definitions, call relationships, imports, dependencies, and surrounding architectural context needed for smell analysis and refactoring. This reduces the cognitive and operational burden on the developer, who should not be expected to manually reconstruct project-level context inside a prompt. More importantly, IDE integration enables the tool to act on real code artifacts, which is necessary when moving from conversational suggestions to actionable maintenance support.

A third implication is that **retrieval and context-grounding should be treated as first-class design elements rather than optional enhancements**. The reviewed studies consistently suggest that repository-aware workflows are more reliable than non-grounded prompting when the task depends on broader structural information. Tool builders should therefore consider retrieval mechanisms that expose relevant project artifacts, prior refactoring examples, structural descriptions, or domain documentation to the model at generation time. In practice, this means using RAG not merely to add more context, but to provide the right context: code elements, dependencies, and examples that are specifically relevant to the smell or refactoring under consideration. This is particularly important for operations such as *Move Method*, *God Class* decomposition, or other transformations that depend on cross-class reasoning.

A fourth implication is that **LLM outputs should never be treated as final actions without validation layers**. Across the reviewed literature, some of the most effective tools were those that embedded the LLM within a broader verification workflow involving compilation checks, static analysis, test execution, or feasibility analysis. This suggests that future tools should act less as direct code generators and more as gatekept assistants whose recommendations are filtered through external correctness checks before being surfaced to the user. The design goal should not be to maximize generation autonomy, but to minimize unsafe or structurally invalid suggestions. In this respect, tools such as MM-ASSIST (Bellur et al., 2025) [P023] and EM-ASSIST (Pomian et al., 2024) [P029] are especially instructive: both combine LLM generation with IDE support, static analysis, ranking, and filtering mechanisms that reduce hallucinations and improve recommendation quality before execution.

A fifth implication is that **iterative refinement workflows are preferable to one-shot interactions**. The reviewed evidence suggests that high-quality refactoring support rarely emerges from a single prompt-response exchange. Instead, more reliable systems decompose the task into stages such as smell identification, candidate generation, validation, critique, ranking, and, when needed, regeneration. This means that future tools should be designed around explicit workflows that support refinement rather than around single-turn completion interfaces. Multi-step designs can also make the behavior of the system more intelligible to developers, since intermediate reasoning products — such as identified smells, rejected candidates, validation failures, or ranking signals — can be exposed and inspected.

A sixth implication is that **developer trust depends on visible human control and transparency**. The reviewed studies suggest that developers are more willing to use LLM-based refactoring tools when they remain in control of the decision process and when the tool makes its suggestions easier to inspect, revise, and justify. Future tools should therefore avoid presenting refactorings as opaque, authoritative answers. Instead, they should make clear which smell was identified, which context was retrieved, which checks were performed, how confident the system is, and why a given recommendation was ranked above alternatives. Trust is unlikely to increase through automation alone; rather, it is strengthened when the system makes its limitations and decision path visible, allowing developers to exercise calibrated judgment.

A seventh implication is that **tool interfaces should support smell-specific and intention-aware interaction**. Generic requests such as "refactor this code" place too much inferential burden on the model and tend to produce weaker or more ambiguous results. The literature suggests that response quality improves when the request is framed in terms of specific smells, quality goals, or transformation intentions. Tool interfaces should therefore allow developers to invoke refactoring support via explicit smell names, quality attributes, or structured commands, rather than relying entirely on open-ended natural-language prompts. This design choice can improve precision while also helping standardize interaction around known maintenance concepts.

Finally, the literature suggests that **the most promising future for LLM-assisted refactoring lies in hybrid, collaborative, and model-agnostic tools**. Rather than replacing developer judgment, effective tools appear to be those that combine LLM reasoning with repository context, IDE-native capabilities, external validation, and human oversight. The practical lesson is that usefulness does not come from generation alone, but from orchestration. Future tool design should therefore prioritize systems that are easy to integrate, operationally transparent, grounded in the project context, and explicitly structured to keep the developer in control.

6 Consolidated Assessment of Effectiveness, Reliability, and Practical Usefulness

The Introduction framed an important gap in the literature around three related but distinct properties of LLM-based code smell detection and refactoring: effectiveness, reliability, and practical usefulness. The findings synthesized in Section 4 and the implications derived in Section 5 allow these dimensions to be revisited explicitly. For the purpose of this synthesis, effectiveness refers to the extent to which an approach succeeds in performing the intended detection or refactoring task under the reported evaluation setting; reliability concerns whether the resulting outputs can be depended upon to remain structurally and behaviorally valid; and practical usefulness concerns whether such capabilities can be integrated into developer workflows in a usable, transparent, and controllable manner.

Effectiveness. The evidence supports a conditional rather than universal view of effectiveness. Prompt-based LLMs can successfully contribute to both code smell detection and refactoring, but their performance depends strongly on task complexity and available context. *Findings #1–#3* show that standalone prompting is less effective when tasks require inter-class or repository-level reasoning, whereas hybrid and retrieval-augmented approaches are more frequently associated with stronger detection metrics, test-passing outcomes, and structurally appropriate transformations. Therefore, the literature does not support the conclusion that prompt-based LLMs are uniformly effective in isolation; rather, effectiveness improves when the model is supplied with task-relevant structural and repository context. Because evaluation settings and metrics remain heterogeneous, the available evidence does not support a single general effectiveness estimate across tasks.

Effectiveness:

Prompt-based LLMs are effective in some code smell detection and refactoring settings, but effectiveness is strongly context-dependent. The strongest evidence favors hybrid and repository-aware configurations over unconstrained standalone prompting, particularly for structurally demanding tasks.

Reliability. Reliability is more strongly associated with the mechanisms surrounding the LLM than with raw generation capability alone. *Findings #2–#4* indicate that repository grounding, static analysis, compilation or test-based validation, feasibility checks, and feedback-driven regeneration repeatedly reduce the risk of hallucinated, structurally invalid, or behaviorally unsafe outputs. The reviewed evidence therefore suggests that raw LLM responses should not generally be treated as reliable final decisions or transformations. Reliability emerges when generation is embedded within a controlled workflow capable of constraining, checking, rejecting, and, where appropriate, regenerating model outputs.

Reliability:

Prompt-based LLM output is not sufficiently reliable by itself for high-confidence maintenance tasks. Reliability is substantially strengthened when generation is grounded in project context and coupled with explicit validation and iterative correction mechanisms.

Practical usefulness. The evidence indicates that practical usefulness depends on more than technical performance. *Finding #6* shows that developers tend to value LLMs as interactive and revisable assistants while remaining cautious about fully autonomous code changes. *Finding #5* provides a complementary perspective by showing that benefits also increase when users remain actively engaged in reasoning and evaluation rather than delegating the task entirely to the model. Together with the implications discussed in Section 5, these results suggest that practical usefulness is greatest when LLM support is integrated into existing development environments, exposes relevant context and validation results, supports iterative interaction, and preserves visible human control over final decisions.

Practical Usefulness:

LLM-based maintenance support appears most practically useful when it augments rather than replaces developer judgment. Transparent, iterative, IDE-integrated, and human-supervised workflows are better aligned with developer needs than autonomous one-shot generation.

Taken together, these dimensions should not be treated as interchangeable. An approach may demonstrate effectiveness in completing a benchmark task without being sufficiently reliable for unsupervised use, while technically validated outputs may still provide limited practical value if they

are difficult to inspect, integrate, or control. The reviewed evidence therefore supports a layered interpretation: effectiveness depends primarily on task and context alignment; reliability depends on grounding, validation, and orchestration; and practical usefulness depends on how these technical capabilities are embedded within transparent and human-supervised workflows. The strongest overall pattern in the corpus is thus not the superiority of LLM generation alone, but the value of combining prompt-based models with contextual information, external verification, iterative interaction, and developer oversight.

7 Threats to Validity

This section discusses the main threats to validity associated with this systematic literature review and the measures adopted to mitigate them. As in any secondary study, the validity of the findings depends not only on the quality of the primary studies, but also on the design of the review protocol, the operationalization of the selection criteria, and the interpretation of heterogeneous evidence.

Construct Validity: Construct validity concerns whether the review protocol and data extraction procedures accurately captured the phenomenon under investigation, namely the prompt-based use of LLMs for code smell detection and refactoring.

A primary threat arises from conceptual ambiguity in the primary literature. Terms such as refactoring, code transformation, code migration, and code restructuring are not always used consistently. This is problematic because some studies adopt a broad notion of refactoring that does not align with the stricter definition used in this review, *i.e.,* behavior-preserving restructuring of existing code. Consequently, borderline studies may be inadvertently retrieved or provisionally retained unless they are carefully examined during the screening and full-text assessment phases, as was done in this study.

A second threat concerns the operationalization of the inclusion and exclusion criteria. The review intentionally excluded studies centered on fine-tuning, retraining, or specialized adapted models in order to preserve a clear focus on prompt-based LLM use. However, in some hybrid systems, the boundary between prompt-based interaction and model adaptation was not immediately explicit, creating room for interpretive judgment during the eligibility assessment. To mitigate this issue, both authors sought to reach an agreement on inclusion decisions, thereby reducing potential bias arising from subjective interpretation.

A third threat relates to incomplete reporting in the primary studies. In some cases, key variables such as the exact LLM family, target programming language, or evaluation setting were not fully specified. This limits the precision of descriptive synthesis, even when the study is clearly relevant to the review scope. To mitigate this issue, such studies were retained when eligibility was clear, while missing attributes were recorded as Not Reported.

A fourth threat concerns the hierarchical grouping and subcategorization of studies. This grouping was introduced to facilitate synthesis across heterogeneous contributions, but some papers could plausibly fit more than one group. This risk was limited because the grouping was used only to support organization and narrative synthesis; no core conclusion of the review depended directly on group membership as an evaluative variable.

Conclusion Validity: Conclusion validity concerns whether the conclusions of the review are justified by the evidence provided in the primary studies.

The main threat is the heterogeneity of the corpus. The included studies differ substantially in terms of models, tasks, datasets, programming languages, smells, refactoring types, and evaluation metrics. As a result, cross-study comparisons are necessarily limited, and differences reported in the literature may reflect experimental context rather than general superiority of one approach over another.

For the same reason, this review did not perform a formal meta-analysis. The primary studies do not report outcomes in a sufficiently uniform way to support statistical aggregation into a single combined effect estimate. Instead, the review synthesizes the literature by identifying recurring tendencies and analytically grounded patterns across studies. This means that the conclusions should be interpreted as evidence-informed trends in the literature, not as precise quantitative estimates of how much one model, prompting strategy, or architecture outperforms another.

A further threat is the rapid evolution of the field itself. Because LLM capabilities, tools, and publication practices change quickly, some findings may be time-sensitive. To mitigate this risk, the review adopts a cautious interpretive stance and avoids strong universal claims where the evidence remains heterogeneous or immature.

Another threat concerns the methodological quality of the primary studies. This review did not conduct a formal quality appraisal or assign methodological quality scores to the included studies; eligibility was determined primarily according to topical relevance, methodological scope, and the predefined inclusion and exclusion criteria. Consequently, studies with weaker empirical rigor, incomplete reporting, or limited methodological quality may have contributed to the synthesized evidence and could therefore affect the robustness of some conclusions. This risk is partially mitigated by the fact that the search was conducted across established scientific databases and digital libraries, including IEEE Xplore, ACM Digital Library, Scopus, ScienceDirect, and Wiley, and that a substantial portion of the corpus was published in established software engineering venues such as ICSE, SANER, ICSME, ICPC, MSR, and ASE, as shown in Table 5. Nevertheless, indexing or publication venue alone cannot guarantee the methodological quality of an individual study. For this reason, the conclusions of this review should be interpreted as reflecting the evidence reported in the selected literature rather than as being weighted by an independent assessment of study quality.

Internal Validity: Internal validity concerns the credibility of the review process itself and the extent to which the final corpus may have been affected by selection or reconciliation bias.

One important threat is that screening and final inclusion decisions were not fully performed by two independent hu-

man reviewers across the entire corpus. Although the review followed a structured protocol, this may still have introduced subjectivity in borderline cases. To mitigate this risk, seven studies initially marked as MAYBE by the first author during manual screening were re-examined by the second author based on title and abstract analysis, providing an additional human assessment for ambiguous cases. However, as this secondary review was limited to a subset of studies rather than the full screening process, it should be interpreted as a partial mitigation rather than a full dual-review screening. Because the screening design did not involve two independent human reviewers assessing a common set of studies across the full selection process, no inter-rater reliability coefficient, such as Cohen's Kappa, was calculated. This represents an additional limitation of the selection process, and the NotebookLM-supported second opinion should not be interpreted as equivalent to independent human inter-rater agreement.

A related threat concerns the use of NotebookLM during screening. In the primary AI-assisted selection phase, NotebookLM was applied to full texts and used as a source-grounded support mechanism for discrepancy analysis. In the snowballing phase, however, the tool was used differently: it screened titles and abstracts rather than full papers. This difference could have caused some relevant studies to be deprioritized during snowballing. However, this risk is limited because abstract-based screening is consistent with standard snowballing practice in preliminary filtering, especially when the candidate pool is large, and all promising cases were still subjected to manual validation before inclusion.

Another internal threat lies in the behavior of NotebookLM itself. As observed during reconciliation, the tool improved sensitivity by surfacing potentially relevant studies, but it also showed semantic over-inclusion in some cases, particularly around broad uses of the term refactoring. This risk was mitigated by conducting manual and AI-assisted screening separately, treating AI outputs as evidence-supported suggestions rather than final decisions, and preserving human responsibility for inclusion.

Further threats arise from excluding papers whose full text could not be retrieved and from removing some studies only after full-text verification and version consolidation. Although methodologically justified, these decisions may still have introduced availability bias or reflected eligibility problems that were only detectable at later stages of the review.

External Validity: External validity concerns the extent to which the findings of this review can be generalized beyond the analyzed corpus.

A first limitation is that the literature itself is concentrated around a small number of model families, especially OpenAI models, and around a relatively narrow set of programming languages, particularly Java and other object-oriented contexts. As a result, the findings are more readily generalizable to mainstream LLM APIs and object-oriented codebases than to underrepresented ecosystems such as functional languages or domain-specific languages.

A second limitation stems from the review scope. This study intentionally excluded work centered on fine-tuned or retrained models, as well as studies whose primary focus was outside code smell detection and refactoring. This choice preserved conceptual focus, but it also limits generalization to the broader landscape of LLM-based software maintenance.

A third threat concerns publication, indexing, and search-term sensitivity. The search strategy was intentionally recall-oriented, combining broader technological expressions with more specific terminology to provide complementary retrieval paths. This design involves an inherent trade-off: broader expressions may increase the retrieval of studies outside the intended scope, whereas any finite combination of search terms may still fail to capture all terminological variations used to describe the concepts addressed by the research questions. Potential over-retrieval was handled during the subsequent eligibility assessment through the predefined inclusion and exclusion criteria. At the same time, complete retrieval cannot be guaranteed, since alternative formulations of the search terms may produce slightly different result sets. After the original search, study selection, and evidence synthesis had been completed, we identified potential variations in the formulation of some search terms and conducted additional sensitivity checks to assess the robustness of the retrieval strategy. These checks considered alternative term formulations and the presence or absence of model-specific expressions. The resulting sets were largely consistent with the original retrieval: most additional records had either already been retrieved through another database or did not satisfy the predefined eligibility criteria. However, one additional eligible study that was not present in the original study pool was identified. Because these checks were conducted after completion of the predefined systematic search process, the study was not retrospectively incorporated into the evidence corpus. Selectively extending the corpus based on individual post hoc query variations would require reopening the search protocol and systematically reapplying a revised strategy across all databases and subsequent review stages. We therefore interpret this result as empirical evidence of residual search-term sensitivity rather than as an extension of the original corpus. Although the tested variations produced results very close to those of the original strategy, complete retrieval cannot be guaranteed, and some relevant studies may remain outside the analyzed corpus. Because the automated search was conducted on November 5, 2025, studies first made publicly available after this date fall outside the review window and may therefore not be represented in the corpus.

A fourth limitation concerns the use of backward snowballing without a complementary forward-snowballing stage. Backward snowballing was adopted to extend the database search by tracing relevant antecedent studies cited by the selected papers. However, forward snowballing could have identified more recent studies that subsequently cited papers in the corpus, which is particularly relevant in a rapidly evolving research area. Its absence therefore introduces a residual risk that some newer follow-up studies were not captured. At the same time, forward citation tracking would require an additional citation-indexing cutoff, since citing studies continue to accumulate after the original search date. The findings should therefore be interpreted as representing the literature identifiable through the predefined database-search and backward-snowballing procedure within the review window, rather

than as guaranteeing exhaustive coverage of all subsequent work.

8 Conclusion and Future Work

This paper presented a systematic literature review on the prompt-based use of frozen-parameter LLMs for code smell detection and code refactoring. The review showed that this is a recent and rapidly expanding research area, but still one marked by methodological heterogeneity, uneven evaluation practices, and fragmented evidence across technical, methodological, educational, and human-centered perspectives. This emerging character is also reflected in the publication landscape: although the topic has already appeared in established software engineering venues such as ICSE, SANER, ICSME, ASE, MSR, and ICPC, the corpus remains widely dispersed across many outlets and includes a visible share of preprints, suggesting a field that is gaining recognition while still consolidating its research agenda. Despite this diversity, some recurring patterns emerged clearly from the analyzed studies: standalone prompting tends to be less reliable in structurally demanding scenarios; repository-aware and hybrid approaches are often associated with better outcomes; and more dependable workflows usually combine task decomposition, contextual grounding, static analysis, testing, and human supervision. Beyond consolidating these technical findings, this study also showed that the practical usefulness of LLM-based support depends not only on model capability, but also on interaction design, validation mechanisms, and developer trust. Methodologically, the review further indicated that source-grounded AI assistance can be incorporated into the study selection process as a support mechanism for discrepancy analysis, suggesting a potentially useful direction for future secondary studies when combined with explicit human oversight.

Future work can be pursued along at least two complementary directions. From the perspective of secondary studies, the experience reported here suggests opportunities to refine SLR protocols for LLM-based software maintenance, especially by clarifying task boundaries, reporting prompting and validation settings more explicitly, improving the comparability of evaluation criteria, and conducting broader analyses of agreement, reproducibility, and evidence maturity. In particular, future reviews could build on the methodological lessons of this study by further examining how source-grounded AI support can be used responsibly in screening, reconciliation, and evidence synthesis without reducing human accountability. From the perspective of primary research and tool building, the findings of this review point to a promising agenda centered on the practical implications discussed in Section 5: context-aware IDE integration, retrieval-augmented grounding, validation layers, iterative workflows, and visible developer control. This agenda is particularly relevant to underexplored programming paradigms, as the current literature is concentrated on object-oriented codebases and development contexts and provides limited evidence for functional languages. A natural next step, therefore, is to investigate how the consolidated knowledge identified in this review can be operationalized in existing refactoring environments for such underrepresented ecosystems, including the adaptation of RefactorEX (Pereira et al., 2025) with prompt-based LLM support for code smell detection, refactoring recommendation, and, where appropriate, validated semi-automatic or automatic execution.

Acknowledgements

This research was partially supported by the Brazilian funding agency CNPq through a PIBIC/CNPq grant (2025–2026).

Data Availability Statement

The data supporting the findings of this study are openly available in our replication package hosted on Zenodo: `https://doi.org/10.5281/zenodo.19368366`.

References

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. N., and Dig, D. (2020). 30 years of software refactoring research: a systematic literature review. *ArXiv*, abs/2007.02194:1–23.

Akalanka, I., Silva, S. D., Ganeshalingam, M., Abeykoon, A., Wijendra, D., and Krishara, J. (2025). AI powered integrated code repository analyzer for efficient developer workflow. In *8th International Research Conference on Smart Computing and Systems Engineering (SCSE)*. Institute of Electrical and Electronics Engineers Inc.

Alomar, E. A., Venkatakrishnan, A., Mkaouer, M. W., Newman, C. D., and Ouni, A. (2024). How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *21st IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 202–206. Institute of Electrical and Electronics Engineers Inc.

Alomari, N., Redah, M., Ashraf, A., and Alshayeb, M. (2026). Using LLMs to enhance code quality: A systematic literature review. *Information and Software Technology*, 190:107960.

Bellur, A. and Batole, F. (2025). IDE native, foundation model based agents for software refactoring. In *2nd IEEE/ACM IDE Workshop (IDE)*, pages 42–45. Institute of Electrical and Electronics Engineers Inc.

Bellur, A., Batole, F., Ullah, M. R., Dilhara, M., Zharov, Y., Bryksin, T., Ishikawa, K., Chen, H., Morimoto, M., Hosomi, T., Nguyen, T. N., Rajan, H., Tsantalis, N., and Dig, D. (2025). Together we are better: LLM, IDE and semantic embedding to assist move method refactoring. In *41st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 430–442. Institute of Electrical and Electronics Engineers Inc.

Borg, M. (2025). Trust calibration in IDEs: Paving the way for widespread adoption of AI refactoring. In *2nd IEEE/ACM IDE Workshop (IDE)*, pages 37–41. Institute of Electrical and Electronics Engineers Inc.

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583.

Cordeiro, J., Noei, S., and Zou, Y. (2024). An empirical study on the code refactoring capability of large language models. arXiv preprint arXiv:2411.02320.

Cordeiro, J., Noei, S., and Zou, Y. (2026). An empirical study on the code refactoring capability of large language models. *ACM Trans. Softw. Eng. Methodol.*

Das, J. K., Mondal, S., and Roy, C. K. (2025). Why do developers engage with ChatGPT in issue-tracker? Investigating usage and reliance on ChatGPT-generated code. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 68–79. Institute of Electrical and Electronics Engineers Inc.

Deo, S., Hinge, D., Chavan, O. S., Wang, Y. O., and Mkaouer, M. W. (2024). Analyzing developer-ChatGPT conversations for software refactoring: An exploratory study. In *21st IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 207–211. Institute of Electrical and Electronics Engineers Inc.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*.

Dilhara, M., Bellur, A., Bryksin, T., and Dig, D. (2024). Unprecedented code change automation: The fusion of LLMs and transformation by example. *Proceedings of the ACM on Software Engineering*, 1:631–653.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *45th IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. Institute of Electrical and Electronics Engineers Inc.

Felizardo, K. R., Deizepe, A., Coutinho, D., Gomes, G., Meireles, M., Gerosa, M., and Steinmacher, I. (2025). On the difficulties of conducting and replicating systematic literature reviews studies using LLMs in software engineering. In *2nd IEEE/ACM International Workshop on Methodological Issues with Empirical Studies in Software Engineering (WSESE)*, pages 20–23.

Felizardo, K. R., Lima, M. S., Deizepe, A., Conte, T. U., and Steinmacher, I. (2024). ChatGPT application in systematic literature reviews in software engineering: an evaluation of its accuracy to support the selection activity. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 25–36.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley, 1 edition.

Guo, Q., Xie, X., Liu, S., Hu, M., Li, X., and Bu, L. (2025). Intention is all you need: Refining your code from your intention.

Habib, M. B., Jannat, A. T., and Azhar, T. (2025). Transformer-based multi-label classification for code smell detection in software engineering. In *2025 6th International Conference for Emerging Technology (INCET)*, pages 1–6.

Jin, H., Sun, Z., and Chen, H. (2024). RGD: Multi-LLM based agent debugger via refinement and generation guidance. In *2024 IEEE International Conference on Agents (ICA)*, pages 136–141.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: a database of existing faults to enable controlled testing studies for Java programs. pages 437–440.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report, Keele University.

Li, J., Gao, Y., Yang, Y., Bai, Y., Zhou, X., Li, Y., Sun, H., Liu, Y., Si, X., Ye, Y., Wu, Y., Lin, Y., Xu, B., Ren, B., Feng, C., and Huang, H. (2025). Fundamental capabilities and applications of large language models: A survey. *ACM Comput. Surv.*, 58(2).

Martinez, S., Xu, L., Elnaggar, M., and Alomar, E. A. (2026). Software refactoring research with large language models: A systematic literature review. *Journal of Systems and Software*, 235:112762.

Melo, R., Simões, P., Gheyi, R., d'Amorim, M., Ribeiro, M., Soares, G., Almeida, E., and Soares, E. (2026). Agentic LMs: Hunting down test smells. *IEEE Software*, 43(1):32–40.

Menolli, A., Strik, B., and Rodrigues, L. (2024). Teaching refactoring to improve code quality with ChatGPT: An experience report in undergraduate lessons. In *ACM International Conference Proceeding Series*, pages 563–574. Association for Computing Machinery.

Mesbah, D., El Madhoun, N., Agha, K., and Chalouati, H. (2025). *Leveraging Prompt-Based Large Language Models for Code Smell Detection: A Comparative Study on the MLCQ Dataset*, pages 444–454.

Midolo, A. and Di Penta, M. (2025). Automated refactoring of non-idiomatic Python code: A differentiated replication with LLMs. In *33rd IEEE International Conference on Program Comprehension (ICPC)*, pages 113–123. IEEE Computer Society.

Midolo, A. and Tramontana, E. (2025). Refactoring loops in the era of LLMs: A comprehensive study. *Future Internet*, 17(9).

Miño, J., Andrade, R., Torres, J., and Chicaiza, K. (2024). Leveraging generative artificial intelligence for software antipattern detection. In *Communications in Computer and Information Science*, volume 2102 CCIS, pages 138–149. Springer Science and Business Media Deutschland GmbH.

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., and Mian, A. (2025). A comprehensive overview of large language models. *ACM Trans. Intell. Syst. Technol.*, 16(5).

Oliveira, E. C., Keuning, H., and Jeuring, J. (2025). Uncovering behavioral patterns in student–LLM conversations during code refactoring tasks. In *25th Koli Calling International Conference on Computing Education Research*, Koli Calling '25, New York, NY, USA. Association for Computing Machinery.

Pandini, G., Martini, A., Videsjorden, A. N., and Fontana, F. A. (2025). An exploratory study on architectural smell refactoring using large languages models. In *22nd IEEE International Conference on Software Architecture (ICSA-C)*, pages 462–471. Institute of Electrical and Electronics Engineers Inc.

Pereira, G., Vegi, L. F. M., and Di-Iorio, V. (2025). Building refactoring tools: Insights from RefactorEx. In *1st Workshop on Software Engineering for Functional Programming (SE4FP)*, pages 9–14.

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Sokolov, A., Bryksin, T., and Dig, D. (2024). EM-Assist: Safe automated ExtractMethod refactoring with LLMs. In *32nd ACM International Conference on the Foundations of Software Engineering (FSE)*, pages 582–586. Association for Computing Machinery, Inc.

Radford, A. and Narasimhan, K. (2018). Improving language understanding by generative pre-training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., and Chadha, A. (2025). A systematic survey of prompt engineering in large language models: Techniques and applications.

Shor, R., Greene, E. A., Sumberg, L., and Weingrad, A. B. (2025). AI tools in academia: Evaluating NotebookLM as a tool for conducting literature reviews. *Psychiatry*, 0(0):1–10. PMID: 40875632.

Silva, L. L., Silva, J. R. D., Montandon, J. E., Andrade, M., and Valente, M. T. (2024). Detecting code smells using ChatGPT: Initial insights. In *18th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 400–406. IEEE Computer Society.

Sobrinho, E., De Lucia, A., and Maia, M. (2021). A systematic literature review on bad smells—5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering*, 47(1):17–66.

Sonkin, V. and Tudose, C. (2025). Beyond snippet assistance: A workflow-centric framework for end-to-end AI-driven code generation. *Computers*, 14(3).

Sun, S. (2024). Enhancing software design and developer experience via LLMs. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2498–2501.

Tsantalis, N., Ketkar, A., and Dig, D. (2022). RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *31st International Conference on Neural Information Processing Systems (NIPS)*, NIPS'17, pages 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Vo, Q.-H., Dao, H., and Fukuda, K. (2025). Harnessing the power of LLMs for code smell detection in Terraform infrastructure as code. In *IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 533–542.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10.

Wu, D., Mu, F., Shi, L., Guo, Z., Liu, K., Zhuang, W., Zhong, Y., and Zhang, L. (2024). iSMELL: Assembling LLMs with expert toolsets for code smell detection and refactoring. In *39th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 1345–1357. Association for Computing Machinery, Inc.

Xu, Y. (2025). MUARF: Leveraging multi-agent workflows for automated code refactoring. In *47th International Conference on Software Engineering (ICSE)*, pages 226–227. IEEE Computer Society.

Zhu, X., Dong, X., Yu, W., Liang, H., and Kong, B. (2025). Refactored Maskformer: Refactor localization and classification for improved universal image segmentation. *Displays*, 87:102981.