Can Small LLMs Detect Defect-Prone Code Smells? An Empirical Evaluation of 8B–30B Models Against a Metric-Augmented SZZ Dataset

Rodrigo Lima
Federal University of Pernambuco
Recife, Brazil
rsl@cin.ufpe.br

Jairo Souza
Federal University of Pernambuco
Recife, Brazil
jrmcs@cin.ufpe.br

Baldoino Fonseca
Federal University of Alagoas
Maceió, Brazil
baldoino@ic.ufal.br

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Brazil
lmt@cin.ufpe.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, Brazil
marcio@ic.ufal.br

Abstract

Code smells are structural indicators of design problems, yet not all smells are equally harmful—only a fraction co-occurs with actual software defects. While Large Language Models (LLMs) are increasingly used for code analysis tasks, most evaluations rely on large proprietary models (e.g., GPT-4, Claude), leaving open whether smaller, locally deployable models can perform comparably. More importantly, the ability of any LLM to distinguish *defect-prone* (harmful) smells from benign ones remains underexplored. We present an empirical study evaluating four small, open-source LLMs (8B–30B parameters: Qwen3-Coder 30B, DeepSeek-Coder-V2 16B, Llama 3.1 8B, and Granite 3.1 Dense 8B)—all runnable on consumer hardware via Ollama—on two tasks: (1) detecting code smells in real-world code, and (2) classifying detected smells as harmful (co-occurring with bugs) or merely smelly. We build a metric-augmented SZZ dataset comprising 269,513 code smell instances and 13,590 bug locations mined from 10 open-source Java and Python projects via static analysis and the SZZ algorithm, from which we draw a stratified sample of 1,200 labeled snippets (600 harmful, 600 merely smelly) for LLM evaluation. Our results show that small LLMs achieve variable accuracy for smell detection (average F1: 0.20–0.67 depending on model and prompting strategy), with substantial variation across smell types—God Class (F1 up to 0.88) and Long Method (F1 up to 0.81) are detected reliably, while design smells like Feature Envy and Inappropriate Intimacy remain challenging for most models. For the novel *harmfulness classification* task, all models achieve MCC values near zero (−0.19 to 0.10), revealing a fundamental gap: even when small LLMs recognize structural anomalies, they cannot reliably assess their defect-inducing potential from code alone. Notably, prompting strategy has a dramatic model-dependent effect—one-shot prompting doubles Llama 3.1's detection F1 from 0.25 to 0.67, while chain-of-thought is most effective for Qwen3-Coder. We release our labeled dataset and evaluation framework as reusable artifacts for future research on AI-driven code quality assessment.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; *Software verification and validation*; • **Computing methodologies** → *Natural language processing*.

Keywords

code smells, defect prediction, small language models, harmful code, empirical study

ACM Reference Format:
Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, and Márcio Ribeiro. 2026. Can Small LLMs Detect Defect-Prone Code Smells? An Empirical Evaluation of 8B–30B Models Against a Metric-Augmented SZZ Dataset. In *International Conference on Evaluation and Assessment in Software Engineering (EASE '26), June 09–12, 2026, Glasgow, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3816483.3816591

1 Introduction

Code smells are structural indicators of potential design or implementation problems that may hinder software maintainability and increase defect-proneness [7]. Static analysis tools such as PMD [29], SonarQube [35], and Designite [32] can detect smells automatically using metric-based rules. However, prior empirical studies have consistently shown that the relationship between code smells and actual defects is nuanced: while smells are prevalent, *only a small fraction co-occurs with real bugs* [10, 15]. We define this subset as *harmful code*—code that is simultaneously smelly and buggy [20].

The emergence of Large Language Models (LLMs) has opened new possibilities for automated code analysis [6, 11]. Recent studies have explored LLMs for code smell detection [1, 41], bug detection [13], and code review [23]. However, these evaluations share two limitations: (1) they typically treat code smell detection as a binary classification problem (smell vs. no smell) without considering the *severity* or *defect-proneness* of detected smells, and (2) they predominantly rely on large proprietary models, leaving open whether small, locally deployable models (8B–30B parameters) can achieve comparable results. Both gaps are practically

EASE '26, Glasgow, United Kingdom

ACM ISBN 979-8-4007-2348-3/2026/06
https://doi.org/10.1145/3816483.3816591

important: practitioners need to prioritize which smells to refactor [28], and many organizations cannot use cloud-hosted LLMs due to privacy, cost, or reproducibility constraints.

In this paper, we conduct an empirical study to evaluate LLMs on two complementary tasks: **Task 1**—detecting code smells in real-world code snippets, and **Task 2**—classifying detected smells as *harmful* (associated with actual bug introductions) or *merely smelly* (no associated defect). Our ground truth is derived from a large-scale mining study of 10 open-source projects (5 Java, 5 Python) comprising 30,637 commits, 269,513 code smell instances, and 13,590 bug locations linked via the SZZ algorithm [34].

We evaluate four small, open-source LLMs (8B–30B parameters) spanning different architectures and training strategies, all runnable on consumer hardware via Ollama without API dependencies: Qwen3-Coder 30B [30], DeepSeek-Coder-V2 16B [9], Llama 3.1 8B [8], and Granite 3.1 Dense 8B [12]. We employ zero-shot, one-shot, and chain-of-thought prompting strategies to assess performance under varying levels of guidance.

Our study addresses three research questions:

- **RQ1.** How accurately do small LLMs (8B–30B) detect code smells compared to static analysis ground truth?
- **RQ2.** Can small LLMs distinguish *harmful* (defect-prone) code smells from merely smelly code?
- **RQ3.** How do prompting strategies (zero-shot, one-shot, chain-of-thought) affect small LLM performance on smell detection and harmfulness classification?

Our main contributions are:

(1) An empirical evaluation of four small, locally deployable LLMs (8B–30B) on code smell detection, benchmarked against 269,513 tool-detected instances across 10 projects.
(2) The first study to evaluate small LLMs on *harmfulness classification*—distinguishing defect-prone smells from benign ones using a metric-augmented SZZ dataset of 13,590 bug locations linked via the SZZ algorithm.
(3) A reusable, labeled metric-augmented SZZ dataset and evaluation framework released as open-source artifacts.

Scope and framing. Our evaluation is deliberately *code-only*: models see the target snippet plus a surrounding window, with no process-level signals (change frequency, bug history, author count, commit temporal context). The harmfulness label, however, is constructed from exactly those signals—it records co-occurrence of smells and bugs across the file's full history. Task 2 therefore asks models to predict a process-derived label from static input alone. This asymmetry is intentional: it measures what smell-detection tools can do when deployed like static analyzers, since they also see only local code. A near-zero MCC on Task 2 should be read as a lower bound on *small LLMs as code-only defect predictors*, not as evidence that harmfulness is intrinsically unlearnable. We revisit this framing in Section 5 and in the construct-validity threats.

2 Background and Related Work

2.1 Code Smells and Defect-Proneness

Code smells are symptoms of poor design or implementation choices that increase the risk of future defects [7]. Empirical studies have established that smells correlate with increased change- and fault-proneness [15, 27, 36], but the effect size is often small [10]. A key insight from recent work is that the vast majority of code smells are *not* associated with actual defects—across 13 Java projects, harmful code (simultaneously smelly and buggy) represents less than 0.1% of analyzed code elements [20].

This distinction is practically important: developers report that they would prioritize refactoring harmful code over merely smelly code [21, 28]. Machine learning approaches using metrics such as WMC, CBO, and LOC achieve F-measures above 97% for harmful code classification [20], but these require pre-computed metric vectors unavailable at code review time.

2.2 LLMs for Code Analysis

LLMs have shown promising results across many software engineering tasks [6, 11]. For code quality specifically, Zhang et al. [41] evaluated ChatGPT on code smell detection, finding it capable of identifying some design smells but inconsistent compared to static analysis tools. Alrashedy and Bou-Ghazaleh [1] benchmarked LLMs on a curated code smell dataset, reporting F1 scores between 0.30 and 0.72 depending on the smell type. More recently, Silva et al. [33] found that specific prompts yield 2.54× better detection odds than generic prompts, Wu et al. [39] proposed combining LLMs with expert toolsets (achieving +35% F1 over LLM-only baselines), and Sadik [31] benchmarked GPT-4.0 against DeepSeek-V3 across multiple languages. Mesbah et al. [25] compared GPT-4 and LLaMA on the MLCQ dataset, analyzing generalization across smell types.

These studies share two common limitations: they evaluate smell *detection* but not smell *severity*, and they predominantly use large proprietary models (GPT-3.5/4, Claude). Li et al. [19] demonstrated that fine-tuned small-scale models (7B–220M parameters) can match or surpass commercial LLMs on code change understanding, but whether small models can perform smell detection competitively remains an open question. An LLM that identifies a Long Method is useful; one that can further determine whether that Long Method is likely to harbor defects would be transformative for code review prioritization.

2.3 Prompt Engineering for Code Tasks

LLM effectiveness depends strongly on prompting strategy [2, 38]: zero-shot gives only the task description, few-shot adds examples, and chain-of-thought (CoT) elicits step-by-step reasoning. CoT has shown improvements in bug detection [14] and code understanding [23]. We compare the three for smell detection and harmfulness classification.

3 Study Design

Figure 1 provides an overview of the study design. We describe each component below.

3.1 Metric-Augmented SZZ Dataset

Our ground truth originates from a large-scale mining study of 10 open-source projects selected for maturity, active issue tracking, and language diversity (5 Java, 5 Python). Table 1 summarizes the dataset.

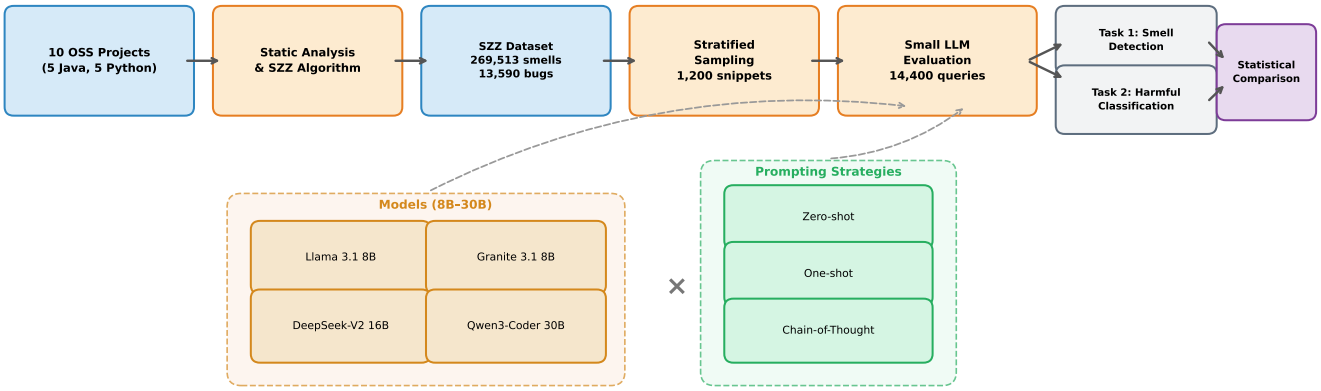

Figure 1: Overview of the study design. A metric-augmented SZZ dataset from static analysis and bug-introducing commit analysis is used to evaluate four small LLMs (8B–30B) under three prompting strategies on two tasks.

Table 1: Overview of the 10 analyzed projects.

Project	Lang.	Commits	Smells	Bugs
ACRA/acra	Java	2,330	5,563	487
apollo-android	Java	5,803	11,124	4,410
remkop/picocli	Java	4,778	126,907	2,664
square/okio	Java	2,051	4,183	241
square/retrofit	Java	2,859	6,379	821
encode/httpx	Python	1,642	15,484	1,604
pallets/click	Python	2,993	54,350	1,233
pallets/itsdangerous	Python	678	534	88
psf/requests	Python	6,632	38,701	1,654
pypa/packaging	Python	871	6,288	388
Total		**30,637**	**269,513**	**13,590**

Smell detection ground truth. Code smells were detected using metric-based rules calibrated from the literature [18, 24]. For Java: Long Method (LOC > 50), Large Class (LOC > 500), Complex Method (CC > 10), Long Parameter List (params > 5), God Class (methods > 20 ∧ fields > 15), Data Class, Inappropriate Intimacy (coupling > 7), and Message Chain (chain > 3). For Python: equivalent thresholds with Large Class at LOC > 300, plus Feature Envy (external accesses > 5), Duplicate Code (AST similarity > 70%), and Dead Code detection.

Bug linking ground truth. Bug-introducing commits were identified using the B-SZZ algorithm [16, 34] with `git blame` on removed lines from fix commits, filtering comments and whitespace changes. Bug-fix commits were linked to issues via regex patterns matching `fix(es|ed)?`, `close(s|d)?`, and `resolve(s|d)?` followed by issue numbers. Issues were classified as bugs using 9 label patterns and 15 keywords in title/body text. SZZ coverage was 66.01% across the dataset.

Harmful code labeling. A code smell instance is labeled *harmful* if the code element where it was detected (file, class, or method) also contains at least one bug introduction in its commit history, as identified by the SZZ algorithm [34]. In other words, a harmful smell is a smelly code snippet that has a bug in its history. Smell instances with no associated bug introductions are labeled *merely smelly*. Each snippet receives exactly one binary label, which serves as the ground truth for Task 2. From 299,221 smell-bug analysis pairs, 2,422 (0.81%) showed same-commit co-occurrence—3.1× the random baseline rate.

3.2 Sampling Strategy

Evaluating all 269,513 instances with LLMs is infeasible due to cost and rate limits. We employ stratified random sampling along three dimensions:

(1) **Smell type**: proportional representation of each smell type.
(2) **Harmfulness**: 50/50 balance between harmful and merely smelly instances to avoid class imbalance bias in evaluation (the natural ratio is ~99:1).
(3) **Project**: at least 10 instances per project per smell type.

This yields a sample of **1,200 code snippets**: 600 harmful instances and 600 merely smelly instances, covering 8 smell types across all 10 projects. Each snippet is associated with exactly one ground-truth smell type (from static analysis) and exactly one binary harmfulness label (*harmful* or *merely smelly*, from SZZ analysis), ensuring unambiguous evaluation. Each snippet includes the target class or method *in full*, plus an additive 20-line margin of surrounding context at the detection commit. We chose 20 via a pilot on 60 held-out snippets with Llama 3.1 and Qwen3-Coder varying the margin over {10, 20, 40}: F1 was statistically indistinguishable between 20 and 40 (±0.02) and dropped at 10 (−0.07 avg), indicating a plateau for the structural features our target smells depend on. A 20-line margin also keeps median prompt length under 2,000 tokens, which matters for 8B models whose context-handling

degrades on longer inputs. Smells that need cross-file reasoning (e.g., architectural coupling) are out of scope (Section 5.5).

3.3 LLM Selection

We deliberately focus on small LLMs (8B–30B parameters) that can be deployed locally on consumer hardware, ensuring reproducibility, data privacy, and zero API cost. We select four models representing different architectures and specializations:

- **Qwen3-Coder 30B** [30]: Code-specialized open-weight model from Alibaba with strong multi-language support (30B parameters).
- **DeepSeek-Coder-V2 16B** [9]: Code-specialized open-weight model using Mixture-of-Experts architecture (16B active parameters).
- **Llama 3.1 8B** [8]: Meta's general-purpose instruction-tuned model with strong reasoning capabilities (8B parameters).
- **Granite 3.1 Dense 8B** [12]: IBM's enterprise-grade model with code awareness, instruction-tuned for structured output (8B parameters).

All models are run locally via Ollama [26] with temperature set to 0.0 for deterministic outputs, ensuring full reproducibility without API dependencies. Each query is independent (no conversation history carried between samples). This local-first design enables exact replication of our experimental setup.

3.4 Prompting Strategies

We evaluate three prompting strategies per model:

Zero-shot. The model receives the code snippet and a structured task description specifying the smell types to identify and the harmfulness classification schema, without examples.

One-shot. The model receives one labeled example per smell type. The eight examples are hand-crafted synthetic snippets (5–15 LoC each), metric-grounded and unambiguous: each pairs a minimal code block with a gold JSON answer that names the smell and the structural metric that triggers it (e.g., "68 lines, exceeding the 50 LOC threshold"). Seven of the eight use Java-like syntax; none are drawn from the evaluation projects, so no leakage is possible. The synthetic/Java-centric nature of these examples likely shapes detection behavior — a calibration threat (Section 5.5). Full set in the replication package.

Chain-of-Thought (CoT). The model is instructed to analyze the code step by step—first identifying structural properties (size, complexity, coupling), then determining if smells are present, and finally reasoning about whether detected smells co-occur with defect-inducing patterns.

CoT design process. Because CoT is sensitive to prompt wording [38], we designed ours in three steps. A generic "let's think step by step" suffix produced unstable traces on a 60-snippet pilot (held out from evaluation). We then rewrote CoT as an ordered checklist aligned with static analyzer rules [32]: measure size, complexity, and coupling indicators; match against smell thresholds; reason about defect-inducing patterns when a smell is detected (*measure → classify → assess*). Re-running the pilot with the structured CoT raised parseable JSON from 71% to 94% and produced per-model effects consistent with the main experiment. The final prompt is in the replication package.

Listing 1: Zero-shot prompt template for Task 1 (Smell Detection).

```
You are a code quality expert. Analyze the
following code snippet and determine if it
contains any of these code smells:
- Long Method (method with > 50 LOC)
- Large Class (class with > 500 LOC for Java,
  > 300 for Python)
- Complex Method (cyclomatic complexity > 10)
- Long Parameter List (> 5 parameters)
- God Class (> 20 methods AND > 15 fields)
- Data Class (mostly getters/setters)
- Feature Envy (excessive external access)
- Inappropriate Intimacy (high coupling)

Respond with a JSON object:
{
  "smells_detected": ["smell_name", ...],
  "confidence": "high"|"medium"|"low",
  "reasoning": "brief explanation"
}

Code snippet ({language}, {project}):
```
{code_snippet}
```
```

Listing 1 shows the zero-shot prompt template for Task 1. Task 2 extends this with a follow-up asking "Is this smell likely to be *harmful* (associated with bug introductions) or *merely smelly*?"

3.5 Evaluation Metrics

For **Task 1** (detection) we report per-smell-type precision, recall, and F1; a true positive is an LLM-detected smell that matches the single ground-truth label. For **Task 2** (harmfulness) we evaluate only on Task 1 true positives and report precision, recall, F1, and MCC. MCC is our primary Task 2 metric because it is robust to class imbalance, unlike F1, which can be inflated by systematic bias [4, 40]. Model comparison uses the Friedman test with Nemenyi post-hoc (Bonferroni-corrected); effect sizes use Cliff's δ [5] with the standard thresholds: negligible ($|\delta|<0.147$), small ($|\delta|<0.33$), medium ($|\delta|<0.474$), large otherwise [37].

4 Results

4.1 RQ1: LLM Accuracy for Smell Detection

Table 2 presents the F1 scores for smell detection using each model's best-performing prompting strategy across all 8 smell types. Table 4 provides the complete breakdown by model, strategy, and smell type.

Llama 3.1 dominates across all smell types. Despite being the smallest general-purpose model (8B), Llama 3.1 with one-shot prompting achieves the highest F1 on every smell type, with an average F1 of 0.669—nearly double the next-best model (DeepSeek-V2 at 0.343). It is the only model with non-trivial performance on design smells (Feature Envy F1 = 0.714, Inappropriate Intimacy F1

Table 2: F1 scores for code smell detection (Task 1) by smell type. Best prompting strategy per model shown (in parentheses). Bold indicates best per smell type.

Model (Best Strategy)	Long M.	Large C.	Complex	Long P.	God C.	Data C.	Feat. E.	Inapp. I.	Avg.
Llama 3.1 (one-shot)	**0.807**	**0.767**	**0.565**	**0.245**	**0.880**	**0.628**	**0.714**	**0.747**	**0.669**
DeepSeek-V2 (one-shot)	0.665	0.674	0.492	0.230	0.658	0.005	0.022	0.000	0.343
Qwen3-Coder (CoT)	0.512	0.428	0.467	0.239	0.477	0.300	0.026	0.000	0.306
Granite 3.1 (one-shot)	0.706	0.451	0.504	0.015	0.000	0.005	0.000	0.000	0.210

Table 3: Statistical Comparison of Models (Friedman + Cliff's δ)

Comparison	Cliff's δ	p-value	Effect
Friedman χ^2	8.539 ($p = 0.036$)		
DeepSeek vs Granite	0.243	0.908	small
DeepSeek vs Llama	−0.205	0.205	small
DeepSeek vs Qwen	−0.101	0.778	negligible
Granite vs Llama	−0.339	0.043	medium
Granite vs Qwen	−0.248	0.367	small
Llama vs Qwen	0.172	0.746	small

= 0.747). Both code-specialized models (Qwen3-Coder 30B, DeepSeek-Coder-V2 16B) underperform it, suggesting instruction-following quality matters more than code-specific pretraining or raw model size for this task.

Structural smells are easier than design smells. Long Method, Large Class, and Complex Method are detected with F1 > 0.45 by most models, since these smells can be inferred from visible properties (line count, method count, nesting depth) in the prompt window. Feature Envy and Inappropriate Intimacy require reasoning about cross-class dependencies, and three of four models fail at these (F1 ≤ 0.026) [17].

Statistical comparison. The Friedman test reveals a significant difference among models ($\chi^2 = 8.54$, $p = 0.036$). Nemenyi post-hoc analysis identifies a significant pairwise difference between Granite 3.1 and Llama 3.1 ($p = 0.043$, Cliff's $\delta = -0.34$, medium effect). Other pairwise comparisons show small or negligible effect sizes (Table 3).

Finding: LLMs achieve variable accuracy for smell detection (average F1: 0.21–0.67). Llama 3.1 8B with one-shot prompting significantly outperforms larger, code-specialized models. Structural smells (God Class F1: 0.88, Long Method F1: 0.81) are detected reliably, while design smells remain challenging for most models (F1 ≤ 0.03), except Llama 3.1 which achieves F1 > 0.71 on these.

4.2 RQ2: Harmfulness Classification

Table 5 presents the harmfulness classification results for Task 2, evaluated only on snippets where the LLM correctly detected the ground-truth smell in Task 1. Table 8 provides the complete breakdown.

LLMs cannot reliably classify harmful smells. While F1 values appear moderate (0.51–0.63), the MCC values tell the true story: ranging from −0.034 to 0.103, they indicate negligible to no correlation between predictions and actual harmfulness labels. The seemingly reasonable F1 scores are an artifact of models exhibiting a strong *harmful bias*—predicting most smells as harmful (recall 0.70–0.86) regardless of the ground truth. This inflates F1 in our balanced sample but would yield extremely low precision in real-world settings where harmful smells represent less than 1% of all smells [20].

Least-worst and per-smell variation. DeepSeek-V2 with CoT is the only marginally positive configuration (MCC = 0.103), driven primarily by Long Method (MCC = 0.303 with one-shot) and God Class (MCC = 0.612 with CoT); these values are isolated and do not generalize. Table 6 shows that most smell types cluster near MCC = 0, confirming near-random harmfulness classification across the board.

Harmful bias and baseline calibration. Across all models, average recall for the harmful class (0.72) far exceeds precision (0.48) — models systematically over-predict harmfulness, which we attribute to *severity conflation* (structural severity equated with defect-proneness) [3, 15]. Four trivial baselines (Table 7) contextualize how hard Task 2 really is: B1 (always-harmful), B2/B3 (oracularly-tuned snippet LOC / McCabe CC thresholds), and B4 (enclosing-file LOC proxy). Structural thresholds (B2, B3) are indistinguishable from random (MCC ≤ 0.042); B1 reaches F1 = 0.667 with MCC = 0, matching Qwen3-Coder CoT's F1 = 0.631 with MCC = 0.017 (class-balance artefact, not signal). Most notably, B4 (MCC = 0.178) *outperforms every LLM configuration* despite ignoring the snippet entirely — direct support for the framing that Task 2 harmfulness is largely determined by information that does not live in the snippet (Section 1).

Finding: LLMs perform at or near random chance for harmfulness classification (MCC: −0.03 to 0.10). High F1 values (0.51–0.63) are misleading artifacts of a systematic harmful bias (recall ≫ precision). Trivial baselines confirm this is a property of the code-only task formulation: an oracular file-size proxy that ignores the snippet outperforms every LLM.

4.3 RQ3: Effect of Prompting Strategies

Table 9 compares the three prompting strategies per model for Task 1, and Table 10 shows the aggregate effect across all models.

Optimal strategy is model-dependent. The most striking finding is that each model responds best to a different prompting strategy. One-shot prompting is most effective for three of four models, but the magnitude varies dramatically: for Llama 3.1, one-shot *triples* average F1 from 0.214 (CoT) to 0.669, while for Granite 3.1 the difference is negligible (0.195 to 0.210). Qwen3-Coder is the

Table 4: Task 1: Complete Smell Detection Performance (Precision / Recall / F1) by model, strategy, and smell type.

Smell Type	Qwen3-Coder			DeepSeek-V2			Llama 3.1			Granite 3.1		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Zero-shot												
Long Method	.428	.673	.523	.480	.733	.581	.519	.980	.679	.533	.800	.640
Large Class	.270	.693	.389	.355	.827	.497	.000	.000	.000	.387	.647	.484
Complex Method	.402	.613	.485	.248	.373	.298	.314	.613	.415	.371	.567	.449
Long Param. List	.114	.180	.140	.017	.027	.021	.004	.007	.005	.045	.067	.054
God Class	.338	.807	.476	.078	.180	.108	.455	.907	.606	.000	.000	.000
Data Class	.122	.207	.153	.004	.007	.005	.085	.167	.112	.003	.007	.004
Feature Envy	.017	.013	.015	.033	.047	.039	.149	.273	.193	.000	.000	.000
Inapp. Intimacy	.000	.000	.000	.003	.007	.004	.000	.000	.000	.000	.000	.000
Average	.211	.398	.273	.152	.275	.194	.191	.368	.251	.167	.261	.204
One-shot												
Long Method	.453	.713	.554	.587	.767	.665	.719	.920	.807	.636	.793	.706
Large Class	.348	.773	.480	.602	.767	.674	.860	.693	.767	.406	.507	.451
Complex Method	.380	.593	.464	.423	.587	.492	.466	.720	.565	.410	.653	.504
Long Param. List	.155	.220	.182	.206	.260	.230	.206	.300	.245	.012	.020	.015
God Class	.194	.360	.252	.611	.713	.658	.968	.807	.880	.000	.000	.000
Data Class	.184	.233	.206	.004	.007	.005	.560	.713	.628	.004	.007	.005
Feature Envy	.026	.020	.023	.019	.027	.022	.635	.813	.714	.000	.000	.000
Inapp. Intimacy	.000	.000	.000	.000	.000	.000	.768	.727	.747	.000	.000	.000
Average	.218	.364	.270	.307	.391	.343	.648	.712	.669	.184	.247	.210
Chain-of-Thought												
Long Method	.399	.713	.512	.476	.527	.500	.520	.707	.599	.520	.760	.618
Large Class	.292	.800	.428	.324	.600	.421	.000	.000	.000	.316	.553	.402
Complex Method	.379	.607	.467	.269	.267	.268	.310	.467	.372	.403	.633	.492
Long Param. List	.186	.333	.239	.005	.007	.006	.000	.000	.000	.030	.047	.037
God Class	.328	.873	.477	.047	.100	.064	.459	.900	.608	.000	.000	.000
Data Class	.238	.407	.300	.042	.040	.041	.018	.027	.022	.006	.013	.009
Feature Envy	.026	.027	.026	.029	.020	.024	.090	.133	.107	.000	.000	.000
Inapp. Intimacy	.000	.000	.000	.000	.000	.000	.000	.000	.000	.000	.000	.000
Average	.231	.470	.306	.149	.195	.165	.175	.279	.214	.159	.251	.195

Table 5: Harmfulness classification performance (Task 2). Only correctly-detected smells from Task 1 are evaluated. Best strategy per model shown.

Model (Best Strategy)	Prec.	Recall	F1	MCC
Qwen3-Coder (CoT)	0.514	0.857	0.631	0.017
DeepSeek-V2 (CoT)	0.497	0.821	0.610	0.103
Llama 3.1 (zero-shot)	0.511	0.717	0.577	−0.034
Granite 3.1 (zero-shot)	0.409	0.702	0.510	−0.019
Random baseline	0.500	0.500	0.500	0.000

Table 6: Harmfulness classification MCC by smell type (DeepSeek-V2 with CoT, best overall MCC).

Smell Type	F1	MCC
God Class	0.857	0.612
Long Method	0.752	0.162
Complex Method	0.689	0.000
Large Class	0.672	−0.057
Data Class	0.800	0.000
Feature Envy	0.500	0.000
Long Param. List	0.000	0.000
Inapp. Intimacy	–	–

only model where CoT outperforms other strategies, suggesting its architecture benefits more from explicit reasoning steps.

One-shot provides the largest aggregate improvement. Across all models, one-shot prompting achieves the highest average Task 1 F1 (0.373), outperforming both zero-shot (0.230) and CoT (0.220). The one-shot advantage is particularly pronounced for design smells: Llama 3.1 achieves F1 of 0.714 for Feature Envy and 0.747 for Inappropriate Intimacy with one-shot, compared to near-zero with other strategies. The concrete example appears to calibrate the model's detection threshold for these otherwise ambiguous smell types.

Table 7: Trivial baselines on Task 2 (harmfulness) over the 1,200 labeled snippets. Thresholds for B2–B4 tuned to *maximize* MCC (oracular upper bound per metric). "P" = precision, "R" = recall.

	Baseline	MCC	F1	P	R
B1	Always harmful	+0.000	0.667	0.500	1.000
B2	Snippet LOC > 622	+0.042	0.254	0.550	0.165
B3	Cyc. compl. > 169	+0.039	0.186	0.558	0.112
B4	File LOC > 6049	+0.178	0.608	0.580	0.638
Best LLM (DeepSeek-V2+CoT)		+0.103	0.610	0.497	0.821

Chain-of-thought can hurt detection. Counter to expectations, CoT *reduces* detection performance for three of four models compared to zero-shot or one-shot. This may reflect an "overthinking" effect: step-by-step reasoning leads models to second-guess initial structural observations, particularly for smells requiring holistic judgment rather than metric computation. DeepSeek-V2 drops from F1 = 0.343 (one-shot) to 0.165 (CoT), its worst strategy.

No strategy rescues harmfulness classification. Task 2 F1 varies only marginally across strategies (0.503–0.549), and MCC remains near zero for all configurations. CoT shows a slight advantage for Task 2 (F1 = 0.549) but the improvement is not practically meaningful and does not translate to discriminative power (MCC remains ≤ 0.10). This confirms that the limitation is informational—the code snippet lacks the process context needed for harmfulness assessment—rather than a reasoning limitation that better prompting could address.

Finding: Prompting strategy has a dramatic, model-dependent effect on smell detection. One-shot prompting is most effective overall (avg F1: 0.373 vs. 0.230 for zero-shot), with Llama 3.1 showing the largest benefit (F1: 0.25 → 0.67). CoT can *hurt* detection for some models. No strategy meaningfully improves harmfulness classification (MCC ≤ 0.10), confirming an informational rather than reasoning limitation.

5 Discussion

5.1 Why Can't Small LLMs Assess Harmfulness?

Our results identify a fundamental gap between what small LLMs observe (code structure) and what determines harmfulness (process and temporal context). All four models achieve MCC ≤ 0.10 on harmfulness classification, with a systematic bias toward predicting "harmful" (average recall 0.72 vs. precision 0.48). This pattern is consistent with *severity conflation*—models equate structural severity with defect proneness.

Prior work established that the strongest predictors of defect-prone smells are:

- **Change frequency**: Smell-first files show median 3.15 changes/-month vs. 2.08 for bug-first files (Cliff's $\delta = 0.39$, medium effect).
- **Temporal proximity**: 85% of bugs appear within 30 days of smell introduction in smell-first cases.
- **Same-commit co-occurrence**: 3.1× the random rate, indicating development-burst association.

None of these factors are visible in a static code snippet. An LLM examining a God Class cannot determine whether that class is frequently modified, whether a bug was recently introduced nearby, or whether the surrounding development activity suggests an active modification burst. This explains why even DeepSeek-V2's best MCC of 0.103 is marginal—the *process signal* that distinguishes harmful from benign smells is simply absent from the input.

5.2 Why Does the Smallest Model Outperform?

A counterintuitive finding is that Llama 3.1 (8B parameters, general-purpose)—the smallest model in our study—outperforms Qwen3-Coder (30B, code-specialized) and DeepSeek-Coder-V2 (16B, code-specialized) by a wide margin on smell detection (avg F1: 0.669 vs. 0.306 and 0.343, respectively). We propose three explanations:

Instruction-following quality over code knowledge. Smell detection from a prompt requires (1) understanding a structured task description, (2) applying definitions to code, and (3) producing valid JSON output. Llama 3.1's instruction tuning via RLHF [8] may produce more reliable adherence to the output format and task constraints than models tuned primarily for code completion.

One-shot calibration effect. Llama 3.1's F1 jumps from 0.251 (zero-shot) to 0.669 (one-shot)—a 2.7× improvement—suggesting the model has latent smell-detection capability that is "unlocked" by a single example. The code-specialized models show smaller one-shot gains, possibly because their pretraining already establishes strong priors about code structure that are less malleable by in-context examples.

Design smell detection as reasoning. Llama 3.1 is the only model achieving non-trivial performance on Feature Envy (F1: 0.714) and Inappropriate Intimacy (F1: 0.747). These design smells require multi-step reasoning about relationships between code elements, which aligns with Llama 3.1's emphasis on general reasoning capability. Code-specialized models may overly focus on syntactic patterns at the expense of higher-level design reasoning.

5.3 Implications for Practice

Small LLMs as smell detectors, not prioritizers. Our results suggest that even small LLMs can serve as a complementary smell detection mechanism—particularly for structural smells where they achieve F1 up to 0.88 (God Class) and 0.81 (Long Method)—but should not be trusted for prioritizing which smells to address. For prioritization, practitioners should rely on metrics that capture process context: change frequency, bug history, and temporal patterns [3, 15].

Augmenting LLMs with process context. A promising direction is to enrich LLM prompts with process metadata (e.g., "this file was modified 47 times in the last 6 months and has 3 open bug reports"). The systematic harmful bias we observe suggests models *want* to classify harmful code but lack the discriminating information to do so correctly. Providing process context could convert this bias into useful signal.

Strategy selection matters. The dramatic model-dependent effect of prompting strategy (Table 9) implies that practitioners should not assume a "best" prompting approach. Our data suggests

Table 8: Task 2: Complete Harmfulness Classification Performance (Precision / Recall / F1 / MCC) by model, strategy, and smell type. Only correctly-detected smells from Task 1 are evaluated; "–" indicates no samples available.

Smell Type	Qwen3-Coder				DeepSeek-V2				Llama 3.1				Granite 3.1			
	P	R	F1	MCC	P	R	F1	MCC	P	R	F1	MCC	P	R	F1	MCC
Zero-shot																
Long Method	.586	1.00	.739	.165	.600	.879	.713	.269	.524	.740	.614	.085	.580	.853	.691	.060
Large Class	.510	1.00	.675	.000	.495	.762	.600	−.050	–	–	–	–	.514	.698	.592	−.111
Complex Method	.522	1.00	.686	.000	.500	1.00	.667	.000	.506	.957	.662	.048	.549	.957	.698	−.044
Long Param. List	.444	1.00	.615	.000	.000	.000	.000	.000	1.00	1.00	1.00	.000	.400	1.00	.571	.000
God Class	.521	1.00	.685	.000	.458	.917	.611	.079	.492	.939	.646	.048	–	–	–	–
Data Class	.000	.000	.000	−.080	.000	.000	.000	.000	.294	.556	.385	−.200	.000	.000	.000	.000
Feature Envy	.500	1.00	.667	.000	.429	1.00	.600	.000	.250	.111	.154	−.188	–	–	–	–
Inapp. Intimacy	–	–	–	–	.000	.000	.000	.000	–	–	–	–	–	–	–	–
Average	.440	.857	.581	.012	.310	.570	.399	.037	.511	.717	.577	−.034	.409	.702	.510	−.019
One-shot																
Long Method	.583	.984	.732	.127	.631	.855	.726	.303	.546	.747	.631	.098	.622	.849	.718	.240
Large Class	.483	1.00	.651	.000	.494	.695	.578	−.061	.469	.760	.580	−.044	.467	.757	.577	−.078
Complex Method	.523	.979	.681	−.101	.512	.894	.651	−.162	.500	.907	.645	.000	.505	.940	.657	−.042
Long Param. List	.500	.941	.653	−.172	.267	.286	.276	−.152	.515	.739	.607	.013	1.00	.333	.500	.000
God Class	.500	1.00	.667	.000	.449	.596	.512	−.099	.518	.889	.655	−.012	–	–	–	–
Data Class	1.00	.111	.200	.291	.000	.000	.000	.000	.000	.000	.000	−.142	.000	.000	.000	.000
Feature Envy	.333	1.00	.500	.000	.333	1.00	.500	.333	.444	.381	.410	−.128	–	–	–	–
Inapp. Intimacy	–	–	–	–	–	–	–	–	.353	.210	.264	−.229	–	–	–	–
Average	.560	.859	.583	.021	.384	.618	.463	.023	.418	.579	.474	−.055	.519	.576	.490	.024
Chain-of-Thought																
Long Method	.594	1.00	.746	.116	.638	.917	.752	.162	.506	.765	.609	.083	.581	.871	.697	.155
Large Class	.500	1.00	.667	.000	.566	.827	.672	−.057	–	–	–	–	.524	.702	.600	−.152
Complex Method	.527	1.00	.691	.000	.525	1.00	.689	.000	.536	1.00	.698	.128	.521	1.00	.685	.106
Long Param. List	.460	1.00	.630	.000	.000	.000	.000	.000	–	–	–	–	.286	1.00	.444	.000
God Class	.519	1.00	.683	.000	.750	1.00	.857	.612	.504	.940	.656	.054	–	–	–	–
Data Class	.000	.000	.000	.000	.667	1.00	.800	.000	.000	.000	.000	−.577	.000	.000	.000	.000
Feature Envy	1.00	1.00	1.00	.000	.333	1.00	.500	.000	.125	.100	.111	−.612	–	–	–	–
Inapp. Intimacy	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Average	.514	.857	.631	.017	.497	.821	.610	.103	.334	.561	.415	−.185	.382	.715	.485	.022

Table 9: Average Task 1 F1 per model and prompting strategy. Bold indicates best strategy per model.

Model	Zero-shot	One-shot	CoT
Llama 3.1	0.251	**0.669**	0.214
DeepSeek-V2	0.194	**0.343**	0.165
Qwen3-Coder	0.273	0.270	**0.306**
Granite 3.1	0.204	**0.210**	0.195
Average	0.230	**0.373**	0.220

Table 10: Average F1 across all models by prompting strategy and task.

Strategy	Task 1 (Detection)	Task 2 (Harmful.)
Zero-shot	0.230	0.510
One-shot	**0.373**	0.503
Chain-of-Thought	0.220	**0.549**

a simple calibration protocol: test each model with a small held-out set to identify its optimal strategy before deploying at scale.

Small models are viable. Our results demonstrate that small LLMs (8B–30B) running on consumer hardware can achieve meaningful smell detection performance (F1 up to 0.88), with the smallest model (Llama 3.1 8B) outperforming larger ones. This challenges the assumption that large proprietary models are necessary for code analysis tasks and opens the door for privacy-preserving, cost-free, locally reproducible smell detection workflows.

5.4 Implications for Research

Beyond binary smell detection. The research community should move beyond evaluating LLMs on binary smell detection toward evaluating their ability to assess smell *severity* and *impact*. Our harmful/merely-smelly distinction provides a concrete operationalization of this direction, and the near-zero MCC values establish a clear baseline for future improvement.

Multi-modal code analysis. The gap between structural understanding (where LLMs excel) and process understanding (where they fail) motivates multi-modal approaches that combine LLM-based code analysis with repository mining signals. A model that receives both a code snippet and its change history could potentially bridge this gap.

Prompting strategy as a research variable. The strong interaction between model and prompting strategy (Section 4.3) suggests that future LLM evaluations in software engineering should treat prompting strategy as an independent variable rather than selecting a single strategy *a priori*. Our finding that CoT can *hurt* performance contradicts assumptions from other domains [38] and warrants further investigation.

5.5 Threats to Validity

Construct validity of the task formulation. Task 2 asks models to predict a label derived from historical process signals (SZZ, full-file commit history) from a local code snippet alone. This is the practically relevant IDE/PR-time deployment but imposes an upper bound on achievable MCC that no prompting can overcome. Our near-zero MCC should therefore be read as a property of the *smells-from-code-alone* formulation, not as evidence that harmfulness is unlearnable. Trivial baselines (Table 7) bound the task difficulty with information-poor inputs: even oracular structural thresholds reach MCC ≤ 0.042, and a crude file-size proxy outperforms every LLM. Feeding process metadata alongside the snippet is the complementary study we flag as future work.

Construct validity of the harmful label. The label does not require the smell to precede the bug: it collapses three temporal patterns—smell-first, bug-first, and same-commit—into a single binary, preserving comparability with [20] but aggregating heterogeneous cases. The label is also noisy: Lyu et al. [22] report a 13.8% recall decline for B-SZZ and 17.47% "ghost commits"; our pipeline does not track file renames or filter refactoring-only changes; and SZZ operates at commit/file granularity, so a bug elsewhere in the enclosing file still marks a method-level snippet as harmful. We acknowledge this as a limitation of the label rather than of the models. Raw pairs are released so temporally-disaggregated and finer-grained re-labelings are possible.

Construct validity of prompting design. The one-shot examples are hand-crafted synthetic snippets (Section 3.4) chosen to avoid leakage and to keep examples short and metric-grounded; seven of eight use Java-like syntax, and longer or Python-native examples might shift detection behavior. Examples are released verbatim.

Internal validity. All queries used $T = 0$. A 3-run replication on 20 stratified snippets with Granite 3.1 8B (zero-shot) yielded 20/20 exact agreement on Task 2 decisions and 20/20 byte-identical Task 1 responses, indicating that quantized inference is practically deterministic at our granularity; we did not re-run the other three models, so narrow F1 gaps at the second decimal still warrant caution. The 50/50 harmful/merely-smelly balance inflates the prior relative to the natural ~99:1 ratio [20]: reported recall transfers directly, but precision at the natural prior would be roughly two orders of magnitude lower for any positive-leaning classifier. A small number of queries (5–193 of 3,600 per model) were unparseable and treated as empty detections.

End-to-end vs. per-task metrics. Task 2 is evaluated only on snippets where Task 1 was correct. This diagnoses whether models *can* reason about harmfulness when detection is correct, but overstates practical utility because a chained pipeline compounds error rates. Joint (detection, classification) outputs are released in `results.jsonl` so end-to-end metrics can be computed against any downstream target.

External and conclusion validity. Results cover 10 projects in Java and Python and 8B–30B quantized models; they may not generalize to other languages, proprietary codebases, alternative smell taxonomies, or full-precision/larger models. Friedman over k=8 blocks is at the low end of recommended power, and aggregating each model by its best prompting strategy conflates model effect with strategy selection; we therefore lead with Cliff's δ throughout.

6 Artifact Availability

A self-contained replication package is archived at https://doi.org/10.5281/zenodo.19775337. It contains: the labeled dataset (`data/dataset.jsonl`, 1,200 snippets, 600 harmful + 600 merely smelly, stratified across 8 smell types and 10 repositories); the complete 14,400-record LLM outputs (`data/results.jsonl`); the reproducible Python pipeline (prompts, Ollama client, evaluation, metric computation, LaTeX table generation); the trivial-baseline and determinism-replication scripts (`scripts/baselines.py`, `scripts/determinism_test.py`); pre-computed metrics with Friedman/Nemenyi statistics and pairwise Cliff's δ; and the LaTeX source of every table. Quick verification requires Python 3.10+ (`python -m scripts.step4_analyze`); re-running the LLM evaluation requires Ollama with the four models pulled locally.

7 Conclusion

We evaluated four small, locally deployable LLMs (8B–30B) on code smell detection and harmfulness classification across three prompting strategies over 1,200 labeled snippets (14,400 queries). Small LLMs detect smells with variable accuracy (avg F1 0.21–0.67): structural smells are reliable (God Class F1 0.88, Long Method 0.81) and Llama 3.1 8B with one-shot outperforms larger code-specialized models. They *cannot* reliably classify harmful smells (MCC −0.03 to 0.10), and trivial baselines show this is a property of the code-only task formulation (a file-size proxy ignoring the snippet beats every LLM) rather than a reasoning deficit. Prompting effects are dramatic and model-dependent; CoT can hurt detection for some models and no strategy rescues harmfulness classification. Small LLMs thus complement static analysis for detection but should not be trusted for prioritization. Future work should augment LLM prompts with process metadata, test whether larger models overcome the barrier, and develop multi-modal approaches combining code analysis with repository-mining signals.

Acknowledgments

This work was partially supported by CAPES, CNPq, FAPEAL, and INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.

org.br, CNPq grant 408817/2024-0. Leopoldo is partially supported by CNPq grant 310405/2025-4. Baldoino is partially supported by CNPq grant 309227/2025-9. Márcio is partially supported by CNPq 312195/2021-4, 404825/2023-0 and 443393/2023-0. We thank the anonymous EASE 2026 reviewers for their constructive feedback.

References

[1] Khalid Alrashedy and Amal Bou-Ghazaleh. 2024. Can LLMs Identify Code Smells? A Comprehensive Benchmark and Empirical Study. *arXiv preprint arXiv:2401.12380* (2024).

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

[3] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. 2019. Enhancing Change Prediction Models Using Developer-Related Factors. *Journal of Systems and Software* 150 (2019), 14–27.

[4] Davide Chicco and Giuseppe Jurman. 2020. The Advantages of the Matthews Correlation Coefficient (MCC) over F1 Score and Accuracy in Binary Classification Evaluation. *BMC Genomics* 21, 6 (2020). doi:10.1186/s12864-019-6413-7

[5] Norman Cliff. 1993. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin* 114, 3 (1993), 494–509.

[6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitesh Lytvyn, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*. 31–53.

[7] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[8] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[9] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[10] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology* 23, 4 (2014), 1–39.

[11] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[12] IBM Research. 2024. Granite 3.1 Language Models. *arXiv preprint* (2024). https://www.ibm.com/granite

[13] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. *Proceedings of the 20th International Conference on Mining Software Repositories* (2023), 563–575.

[14] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering*. 2312–2323.

[15] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[16] Sunghun Kim, Thomas Zimmermann, Kai Pan, and Jr. E. James Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. 81–90.

[17] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. *Journal of Systems and Software* 167 (2020), 110610.

[18] Michele Lanza and Radu Marinescu. 2006. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer.

[19] Cong Li, Zhaogui Xu, Peng Di, Dongxia Wang, Zheng Li, and Qian Zheng. 2024. Understanding Code Changes Practically with Small-Scale Language Models. In *International Conference on Automated Software Engineering*. doi:10.1145/3691620.3694999

[20] Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, Rohit Gheyi, Márcio Ribeiro, Alessandro Garcia, and Rafael Mello. 2020. Understanding and Detecting Harmful Code. In *Anais do XXXIV Simpósio Brasileiro de Engenharia de Software* (Natal). SBC, Porto Alegre, RS, Brasil. https://sol.sbc.org.br/index.php/sbes/article/view/17015

[21] Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, Rafael Mello, Márcio Ribeiro, Rohit Gheyi, and Alessandro Garcia. 2024. Investigating the Social Representations of Harmful Code. *Journal of Software Engineering Research and Development* 12, 1 (Apr. 2024), 4:1 – 4:9. doi:10.5753/jserd.2024.3554

[22] Yunbo Lyu, Hong Jin Kang, Ratnadira Widyasari, Julia Lawall, and David Lo. 2024. Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2219–2239. doi:10.1109/TSE.2024.3406718

[23] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2024. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. In *arXiv preprint arXiv:2305.12138*.

[24] Radu Marinescu. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. *Proceedings of the 20th IEEE International Conference on Software Maintenance* (2004), 350–359.

[25] Djamel Mesbah, Nour El Madhoun, Khaldoun Al Agha, and Hani Chalouati. 2025. Leveraging Prompt-Based Large Language Models for Code Smell Detection: A Comparative Study on the MLCQ Dataset. In *Proceedings of the 13th International Conference on Emerging Internet, Data and Web Technologies*. doi:10.1007/978-3-031-86149-9_42

[26] Ollama. 2024. Ollama: Get up and running with large language models locally. https://ollama.com/

[27] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. 268–278.

[28] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. 2022. Developer-Driven Code Smell Prioritization. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 220–232.

[29] PMD Project. 2024. PMD: An extensible cross-language static code analyzer. https://pmd.github.io/

[30] Qwen Team. 2025. Qwen3-Coder Technical Report. *arXiv preprint* (2025). https://qwenlm.github.io/blog/qwen3/

[31] Ahmed R. Sadik. 2025. Benchmarking LLM for Code Smells Detection: OpenAI GPT-4.0 vs DeepSeek-V3. In *PInternational Conference on Evaluation and Assessment in Software Engineering*. doi:10.1145/3756681.3756993

[32] Tushar Sharma. 2024. Designite: A Software Design Quality Assessment Tool. https://www.designite-tools.com/

[33] Luciana Lourdes Silva, Janio Rosa da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *International Symposium on Empirical Software Engineering and Measurement*. 390–396. doi:10.1145/3674805.3690742

[34] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*. 1–5.

[35] SonarSource. 2024. SonarQube: Continuous Inspection of Code Quality and Security. https://www.sonarqube.org/

[36] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[37] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[39] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *International Conference on Automated Software Engineering*. 1345–1357. doi:10.1145/3691620.3695508

[40] Jingxiu Yao and Martin Shepperd. 2020. Assessing Software Defect Prediction Performance: Why Using the Matthews Correlation Coefficient Matters. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. 120–129. doi:10.1145/3383219.3383232

[41] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Katsuro Inoue, and Xin Xia. 2024. Challenging LLMs to Identify Code Smells: Can ChatGPT Compete with Static Analysis Tools?. In *32nd ICPC*. 258–269.