A Causal Perspective on Measuring, Explaining and Mitigating Smells in *LLM*-Generated Code

Alejandro Velasco
svelascodimate@wm.edu
William & Mary
Williamsburg, Virginia, USA

Daniel Rodriguez-Cardenas
dhrodriguezcar@wm.edu
William & Mary
Williamsburg, Virginia, USA

Dipin Khati
dkhati@wm.edu
William & Mary
Williamsburg, Virginia, USA

David N. Palacio
davidnad@microsoft.com
Microsoft
USA

Luftar Rahman Alif
bsse1120@iit.du.ac.bd
University of Dhaka
Bangladesh

Denys Poshyvanyk
dposhyvanyk@wm.edu
William & Mary
Williamsburg, Virginia, USA

Abstract

Recent advances in large language models (*LLMs*) have accelerated their adoption in software engineering contexts. However, concerns persist about the structural quality of the code they produce. In particular, *LLMs* often replicate poor coding practices, introducing code smells (*i.e.,* patterns that hinder readability, maintainability, or design integrity). Although prior research has examined the detection or repair of smells, we still lack a clear understanding of how and when these issues emerge in generated code.

This paper addresses this gap by systematically ***measuring***, ***explaining*** and ***mitigating*** smell propensity in *LLM*-generated code. We build on the Propensity Smelly Score (*PSC*), a probabilistic metric that estimates the likelihood of generating particular smell types, and establish its robustness as a signal of structural quality. Using *PSC* as an instrument for causal analysis, we identify how generation strategy, model size, model architecture and prompt formulation shape the structural properties of generated code. Our findings show that prompt design and architectural choices play a decisive role in smell propensity and motivate practical mitigation strategies that reduce its occurrence. A user study further demonstrates that *PSC* helps developers interpret model behavior and assess code quality, providing evidence that smell propensity signals can support human judgement. Taken together, our work lays the groundwork for integrating quality-aware assessments into the evaluation and deployment of *LLMs* for code.

CCS Concepts

• **Software and its engineering** → Software verification and validation; • **Computing methodologies** → **Knowledge representation and reasoning**; **Probabilistic reasoning**; • **Theory of computation** → *Semantics and reasoning*.

Keywords

LLMs for code, Code Smells, Interpretable and Trustworthy AI

ACM Reference Format:
Alejandro Velasco, Daniel Rodriguez-Cardenas, Dipin Khati, David N. Palacio, Luftar Rahman Alif, and Denys Poshyvanyk. 2026. A Causal Perspective on Measuring, Explaining and Mitigating Smells in *LLM*-Generated Code. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773164

1 Introduction

LLMs have rapidly moved from experimental tools to everyday assistants in software engineering (SE) [38]. Developers rely on them for many tasks such as code completion [5, 16, 41], summarization [1], program repair [11], clone detection [40] and test generation [39]. As generated code increasingly enters production systems, evaluation must look beyond functional correctness and consider whether the output meets broader standards of maintainable and robust software.

Code smells offer a foundation for evaluating these broader quality concerns. They capture recurring design and implementation issues that affect readability, complexity, and long-term maintainability [8, 23, 36]. Prior work has traditionally attributed the introduction of smells to human-driven factors such as developer experience, time pressure, and project constraints [36]. The increasing involvement of *LLMs* challenges this assumption. Models trained on large-scale code corpora inherit patterns present in their training data, including smelly or suboptimal code [28, 35], and they may introduce additional defects or vulnerabilities during generation [2, 25]. Smell formation now reflects both human design choices and model behavior, which makes traditional explanations incomplete.

Current evaluation practices are not designed to capture this shift. Similarity metrics such as BLEU [24] and CodeBLEU [29], along with execution-based measures, offer useful information about functional correctness but provide limited insight into structural or design quality [20, 32–34]. Although static analysis tools (SATs) can identify smells in model outputs, post-hoc detection alone cannot reveal why a model introduced these issues or which components of the generation process shaped them. As a result, developers lack visibility into the factors that influence the maintainability of *LLM*-generated code.

A deeper understanding of these issues requires analyzing how specific components of the generation pipeline influence both the structural and semantic properties of the output. Viewed from

ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773164

this angle, several key questions emerge for the SE community: *Which elements of the generation process affect the likelihood of smell introduction? How do decoding strategies, model architectures, and prompt formulations shape the structure and maintainability of generated code? Which of these factors can be intervened upon to guide models toward producing higher-quality outputs?*

While these questions remain open, recent work has begun to introduce early indicators of smell propensity in *LLMs*. Velasco et al. proposed the *PSC* metric, a probabilistic score that estimates the likelihood that a model emits tokens associated with a given smell [37]. Introduced as part of a benchmarking effort, *PSC* provides a continuous signal of model tendencies and a promising foundation for investigating the factors behind smell occurrence. Yet the broader dynamics of smell generation remain unexplored. The drivers of variation in *PSC* and their relationship to the structural quality of generated code are still unknown, and the stability of the metric across diverse conditions remains untested. In addition, *PSC* has yet to be used to uncover mechanisms that explain *why* models introduce smells or to support interventions that reduce smell propensity. These gaps hinder a full understanding of how smells emerge in *LLM*-generated code and limit our ability to devise approaches for reducing their occurrence.

Responding to these gaps, we investigate smell generation from a causal perspective. Instead of treating *PSC* as a standalone benchmark metric, we use it as an instrument for reasoning about the factors that influence smell propensity in *LLM*-generated code. Our study begins by assessing the robustness and explanatory value of *PSC*, establishing that it provides a stable foundation for causal analysis. We then quantify the effects of key elements in the generation process, including decoding strategy, model size, model architecture and prompt formulation, on the likelihood of smell occurrence. These causal estimates identify the factors that meaningfully influence smell propensity and guide the design of mitigation strategies that can be applied during inference. Finally, we complement this analysis with a user study that evaluates whether *PSC* helps developers interpret and assess generated code.

Our results show that prompt formulation and model architecture most strongly influence smell propensity, and that targeted adjustments to these factors can significantly reduce smell introduction during inference. We also find that *PSC* captures structural quality signals overlooked by traditional metrics and supports developers in reasoning about generated code. Our specific contributions are listed as follows:

- We validate *PSC* as a stable and informative instrument for analyzing smell propensity.
- We provide a causal analysis quantifying the influence of generation strategy, model size, model architecture and prompt formulation.
- We design prompt-based mitigation strategies that reduce smell propensity during inference.
- We conduct a user study evaluating the practical value of *PSC* for developers.
- We release a dataset, benchmark and implementation to support future research on explaining and mitigating smell generation [14].

2 Motivation & Related Work

Limitations of current evaluations. Existing evaluation methods prioritize surface-level correctness or lexical similarity. As a result, outputs that pass canonical metrics may still exhibit design flaws or maintenance issues. Recent empirical studies have highlighted this gap. Siddiq et al. [32] and Torek et al. [18] showed that both the training data and the code generated from *LLMs* frequently exhibit code smells and insecure constructs. Alif et al. [19] observed that models like Bard and ChatGPT introduce more smells than Copilot or CodeWhisperer in complex tasks. Similarly, Kharma et al. [12] linked outdated training corpora to the persistence of legacy antipatterns. In the context of test generation, Ouédraogo et al. [21] found high rates of test-specific smells in the outputs of GPT-3.5 and GPT-4. Although these findings underscore the need for deeper evaluation, current tools do not provide standardized methods to measure or explain the structural quality of *LLM*-generated code.

Toward principled evaluation and understanding. To address these limitations, recent work has introduced specialized benchmarks and metrics aimed at evaluating and explaining the structural quality of *LLM*-generated code. The *CodeSmellEval* benchmark [37] proposed the Propensity Smelly Score (*PSC*), a probabilistic metric that estimates the likelihood of generating specific types of smell. Zheng et al. [43] developed the RACE framework to incorporate broader quality dimensions such as maintainability and readability. Wu et al. [42] introduced *iSMELL*, a system that combines static analysis with *LLMs* to improve smell detection and refactoring. Although these approaches mark important progress, key questions remain unanswered: *Which generation factors most influence the presence of smell? Can we isolate causal contributors to quality degradation? And can smell generation be systematically reduced through actionable strategies?*

This paper builds upon prior work by shifting the focus from merely detection to explanation and mitigation. We aim to systematically **measure**, **explain**, and **mitigate** code smells in *LLM*-generated code. To support this goal, we adopt the *PSC* as a measurement tool and extend its use across three dimensions.

3 Propensity Smelly Score (*PSC*)

The Propensity Smelly Score (*PSC*) is a probabilistic metric that estimates the likelihood that a *LLM* generates a specific type of code smell [37]. It relies on the next-token probabilities predicted by the model during autoregressive generation. Given a token sequence $w = \{w_1, w_2, \ldots, w_n\}$, the model produces a probability distribution in the vocabulary for each token position. The *PSC* computation process is depicted in Fig. 1.

Let μ denote a code smell instance aligned to a token span (i, j) in w using an alignment function $\delta_\mu(w)$. The *PSC* is computed by aggregating the model's token-level probabilities within this span. For each token w_k in the range $i \le k \le j$, we compute $P(w_k \mid w_1, \ldots, w_{k-1})$ using softmax-transformed logits. The aggregated *PSC* for μ is given by Equation 1:

$$\theta_\mu(w, i, j) = \mathbb{E}_{k=i}^{j} \left[P(w_k \mid w_1, \ldots, w_{k-1}) \right] \tag{1}$$

The function θ_μ reflects the model's average confidence in generating the sequence associated with a given code smell. Higher

PSC values indicate a stronger tendency to produce tokens that correspond to known smells. To support comparisons across models, we adopt a threshold $\lambda = 0.5$ from prior work [37] as a default interpretability criterion, where *a PSC value at or above λ indicates that the model is likely to generate the corresponding smell.* This threshold is used only to interpret the score and does not affect how *PSC* is computed. Adjusting it changes how many instances are labeled as propense but leaves the underlying probabilities unchanged.

3.1 Computation of the Aggregation Function θ

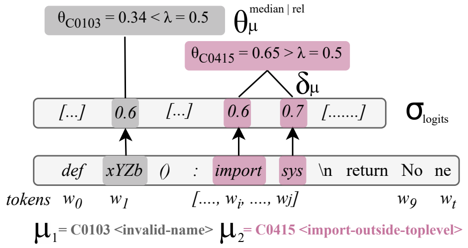

$\theta_{C0103} = 0.34 < \lambda = 0.5$
$\theta_\mu^{\text{median | rel}}$
$\theta_{C0415} = 0.65 > \lambda = 0.5$
δ_μ
[...]
0.6
[...]
0.6
0.7
[.......]
σ_{logits}
def
xYZb
()
:
import
sys
\n
return
No
ne
tokens
w_0
w_1
[...., w_i, ..., wj]
w_9
w_t
μ_1= C0103 <invalid-name>
μ_2= C0415 <import-outside-toplevel>

Figure 1: Propensity Smelly Score (*SCM*) Computation. The python snippet at the bottom contains two code smells: `C0103` (*i.e.,* invalid-name) and `C0415` (*i.e.,* import-outside-toplevel).

The aggregation function $\theta_\mu(w, i, j)$ (Eq. 1) supports multiple computation strategies to estimate *PSC*. Common approaches include taking the mean or median of token-level probabilities within the span associated with the smell. Alternatively, relative aggregation methods can be applied to normalize probabilities based on the surrounding context or the baseline distributions.

Mean and Median Aggregation. The simplest approach aggregates the actual probabilities $P(w_k \mid w_1, \ldots, w_{k-1})$ over the token span (i, j) Two common options are the mean, defined as $\theta_\mu^{\text{mean}} = \frac{1}{j-i+1} \sum_{k=i}^{j} P(w_k \mid w_1, \ldots, w_{k-1})$, and the median, defined as $\theta_\mu^{\text{median}} = \text{median}\left(\{P(w_k \mid w_1, \ldots, w_{k-1}) \mid i \leq k \leq j\}\right)$. The mean captures the overall trend in model confidence but may be influenced by outliers, whereas the median is more robust to local fluctuations and low-confidence predictions.

Relative Aggregation. To account for variability across code smells or token positions, we introduce a normalized variant of the score, called the *relative propensity score*. This version rescales each token probability using empirical bounds observed in a reference dataset:

$$\theta_\mu^{\text{rel}}(w, i, j) = \frac{1}{j-i+1} \sum_{k=i}^{j} \frac{P(w_k) - P_{\min}(w_k)}{P_{\max}(w_k) - P_{\min}(w_k) + \epsilon} \tag{2}$$

Here, $P(w_k)$ denotes the actual probability for token w_k, while $P_{\min}(w_k)$ and $P_{\max}(w_k)$ represent the empirical minimum and maximum values for the same token position within the sample. A small constant ϵ is added to ensure numerical stability. This formulation scales all values to a unit range, facilitating fair comparisons across smells and models.

Implementation. In practice, all three variants of θ are computed from the same set of token-level probabilities. Each reflects a distinct aggregation philosophy: the mean and median quantify absolute confidence, while the relative score captures scale-invariant trends that are useful for comparative evaluations.

4 Methodology

This section presents our methodology for addressing four research questions on measuring, explaining and mitigating code smells in *LLM*-generated code, as well as evaluating the practical utility of *PSC*. We begin by validating *PSC* through robustness and information-gain analyses. We then apply causal inference to identify the factors that shape smell propensity. Building on these insights, we design a mitigation case study and conclude with a user study examining how practitioners interpret and apply *PSC* during code review.

RQ$_1$ [Measure] *How can we measure the propensity of LLMs to generate code smells?* We rely on *PSC* as a probabilistic indicator of smell propensity. To assess its effectiveness, we examine its *robustness* under *SECT* and compare its *information gain* with baselines such as BLEU and CodeBLEU.

RQ$_2$ [Explain] *What factors contribute to the presence of code smells in LLM-generated code?* We examine four intervention scenarios on the behavior of *PSC*: *model architecture*, *model size*, *generation strategy* and *prompt type*. Using causal inference techniques, we estimate the effect of each intervention on the score.

RQ$_3$ [Mitigate] *What strategies can reduce the presence of code smells in code generated LLM?* Based on the insights obtained from **RQ$_2$**, we conduct a case study to demonstrate how controlling these factors can reduce *PSC*, thus lowering the model's tendency to produce smelly code.

RQ$_4$ [Usefulness] *How do developers assess code smells in LLM-generated code when supported by PSC?* We conduct a user study to examine whether *PSC* helps developers evaluate generated code, focusing on its perceived relevance, its impact on confidence and its usefulness in identifying introduced smells.

4.1 Semantic Preserving Code Transformations

To assess the robustness of the *PSC* metric (*i.e.,* **RQ$_1$**) under syntactic variation, we design our methodology based on *Semantic Preserving Code Transformations* (*SECT*). The goal is to evaluate whether *PSC* remains consistent when the input code is modified in ways that preserve functionality but alter its syntax. A reliable quality metric should remain stable under such modifications, as the semantic behavior of the code remains unchanged.

Our methodology draws on prior work in robustness evaluation for neural program repair using natural transformations [15], but we focus specifically on **statement-level transformations**. Broader edits, such as converting while loops to for loops (*While2For*), can alter structural context and influence how smells are detected. Limiting the analysis to statement-level changes preserves semantics and overall structure while still introducing meaningful syntactic variation.

We apply a curated set of six transformations, listed in Table 2, including *Add2Equal*, *SwitchEqualExp*, *InfixDividing*, *SwitchRelation*, *RenameVariable-1* and *RenameVariable-2*. These transformations modify the lexical and syntactic form of code statements without altering control flow or observable behavior. To evaluate robustness, we compute the *PSC* using the **relative aggregation** method (see Eq. 2) for code snippets containing confirmed instances of each smell type. Because raw *PSC* values are bounded and often skewed, we apply a **logit transformation** to improve normality prior to statistical testing.

We conduct a one-way **Analysis of Variance (ANOVA)** to test whether *PSC* scores differ across transformation types, performing this analysis separately for each smell. The results indicate where the metric remains stable under syntactic variation and where it becomes more sensitive to surface-level changes. For each case, we report the F-statistic (F_{anova}), the p-value (p_{value}) and the effect size (η^2), where non-significant p-values and small effects indicate robustness under syntactic changes. To complement these tests, we compute 95% confidence intervals ($CI_{95\%}$) for relative *PSC* scores across transformation variants. Lower variance across intervals suggests stability under syntactic changes, whereas higher variance indicates susceptibility to perturbations.

4.2 Information Gain (*IG*)

To assess how well different metrics explain the presence of severe code smells in generated code (*i.e.,* **RQ**$_1$), we calculate the information gain (*IG*) between each metric and a binary severity label. This enables a direct comparison between *PSC*, which is based on token-level generation probabilities, and existing baselines such as BLEU and CodeBLEU, which rely on similarity to a reference output.

BLEU and CodeBLEU are widely used in code generation benchmarks and are often treated as default baselines. Both rely on surface-level similarity, using n-gram overlap and syntax-based matching to assess how closely a generated snippet aligns with a reference. These metrics are effective for evaluating correctness but are not designed to reflect deeper structural properties such as code smells. Because no established alternatives exist for assessing structural quality in generated code, we include BLEU and CodeBLEU as representative standards for comparison [8, 18]. Their reliance on reference similarity contrasts with *PSC*, which captures likelihood-based signals from the model itself. To compare metrics that draw on such different sources of information, we adopt an information-gain framework that quantifies how much each metric reduces uncertainty about the severity of code smells.

To define severity, we analyze the proportion of smelly tokens in a generated snippet. A snippet is labeled as *high severity* if more than fifty percent of its tokens are flagged as smelly based on static analysis. Otherwise, it is labeled as *low severity.* Let n_s be the number of smelly tokens and n_t the total number of tokens in the snippet. The severity label is assigned according to the following rule:

$$S = \begin{cases} \text{high}, & \text{if } \frac{n_s}{n_t} > 0.5 \\ \text{low}, & \text{otherwise} \end{cases} \tag{3}$$

Let $S \in \{\text{low}, \text{high}\}$ denote the severity label and let X be a continuous score produced by a given metric, such as *PSC*, BLEU or CodeBLEU. The Information Gain is defined as:

$$\text{IG}(S, X) = H(S) - H(S \mid X) \tag{4}$$

Here, $H(S)$ is the entropy of the severity distribution, and $H(S \mid X)$ is the conditional entropy of S given the score X. A lower conditional entropy indicates that the metric provides more information about severity by reducing uncertainty. Therefore, a higher information gain value reflects a stronger relationship between the metric and structural quality.

We expect *PSC* to produce a higher information gain than BLEU and CodeBLEU. Because *PSC* directly captures the model's generation behavior with respect to smelly structures, it should align more closely with severity labels derived from token-level analysis. In contrast, BLEU and CodeBLEU may not reflect such quality differences, especially in cases where syntactically similar outputs differ in maintainability. A higher information gain for *PSC* would confirm its suitability for evaluating design-level concerns in generated code, such as code smells.

4.3 Causal Analysis of Factors Influencing *PSC*

To quantify the causal effect of the intervention settings on *PSC* (see **RQ**$_2$), we construct a structured causal model (*SCM*), illustrated in Fig. 2. Our objective is to estimate the effect of four treatment variables: generation strategy (T_1), model size (T_2), model architecture (T_3), and prompt type (T_4), on two outcomes: relative *PSC* (Y_0) and median *PSC* (Y_1).

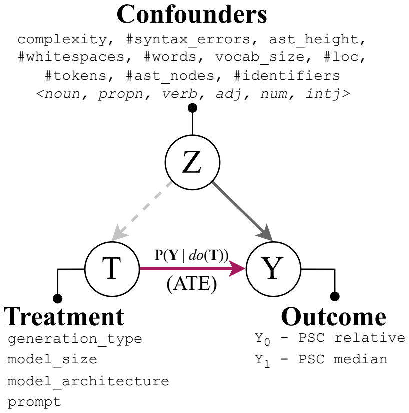


Figure 2: Structural Causal Model (*SCM*)

We adjust for syntactic and lexical features that confound the relationship between treatments and outcomes (*i.e.,* Z). These include lines of code (LOC), token count, number of AST nodes and identifiers, AST height, number of syntax errors and whitespaces, word count and vocabulary size. We also control for the distribution of common part-of-speech (POS) tags in code, such as number of nouns, proper nouns, verbs, adjectives, numerals and interjections.

Previous studies support the inclusion of these variables. Sampling strategies with high temperature tend to produce longer and more deeply nested code, whereas greedy decoding yields shorter and flatter sequences [7]. Model size shapes output as well. Larger models typically produce more concise and focused completions, while smaller ones often generate repetitive or verbose code [4]. Architectural design introduces further variation. Models that use grouped-query attention and sliding windows, such as Mistral, often produce shorter sequences than similarly sized LLaMA models [9]. Prompt formulation has a comparable effect. Detailed prompts, including full function signatures, tend to elicit longer and more structured outputs, whereas vague prompts result in shorter completions [13]. Finally, probabilistic metrics reflect the underlying structure of code. The ASTTrust study [22] shows that metrics such as *PSC* vary with AST complexity and token alignment.

4.3.1 *Computing Average Treatment Effects.*

The average treatment effect (*ATE*), shown in Fig. 2, captures the expected change in *PSC* resulting from an intervention, averaged over the distribution of confounders. Formally, *ATE* is expressed as $P(Y \mid \mathrm{do}(T))$ in Pearl's do-calculus [26]. We identify valid causal estimands by applying graphical criteria (*e.g.,* back-door criterion), which determines the appropriate adjustment set Z from the causal graph.

4.3.2 *Measuring Robustness.*

To assess the reliability of the estimated *ATE* values, we apply four refutation tests. We add random common causes to check for spurious effects, run placebo tests by replacing the treatment with random variables, simulate unobserved confounding to gauge sensitivity to missing factors and perform subset validation to examine stability across data partitions. Consistent patterns across these tests indicate that the estimates are robust and generalizable.

4.3.3 *Interpreting Average Treatment Effects.*

A positive *ATE* indicates that treatment increases *PSC*, suggesting a higher likelihood of generating code smells. In contrast, a negative *ATE* implies a decrease in *PSC*, indicating a potential reduction in the generation of smells. The magnitude of *ATE* reflects the strength of the causal effect. Its credibility is supported by statistical significance and robustness to sensitivity tests.

4.4 *PSC* Mitigation

To address **RQ**$_3$, we investigate whether prompt design can mitigate the generation of code smells as measured by *PSC*. Instead of modifying model parameters or architecture, we focus on inference-time interventions that remain accessible and practical for users.

Our case study builds on the findings of **RQ**$_2$ (specifically T_4), where prompt type emerged as a significant causal factor influencing *PSC*. To operationalize this insight, we isolate the effect of prompt formulation by comparing two strategies: a minimal prompt that presents only a Python function containing a smell, and a structured prompt that provides an explicit task description.

For each smell instance, we generate completions under both prompt conditions using the same underlying model. We then compute the *PSC* for the smelly portion of each generated output and compare the distributions between conditions to assess the impact of prompt design on code smell propensity.

4.5 *PSC* Usefulness

To address **RQ**$_4$, we conducted a user study between subjects in order to evaluate the practical value and perceived usefulness of the *PSC* metric when integrated in the context of code review. This section describes the study design in detail. We also report on how the instrument was validated prior to deployment to ensure clarity and usability.

4.5.1 *Population Profiling.*

The target population for this study consisted of people with experience in software development and familiarity with code generation tools powered by *LLMs*, such as *GitHub Copilot* and *ChatGPT*. Participants were expected to have working knowledge of Python and a general understanding of programming concepts relevant to code structure and maintainability, including the notion of code smells. Although experience with deep learning models (*e.g.,* GPT, T5, or LLaMA) was considered beneficial, it was not required. Our goal was to capture perspectives from academic and industry professionals with practical exposure to *LLM*-assisted programming. Participants were required to be at least 21 years of age. We did not impose restrictions on gender, educational level, or professional background beyond basic software development proficiency. Participation was voluntary and individuals were informed about the purpose of the study and their right to withdraw at any time.

4.5.2 *Sampling.*

We followed a *purpose sampling* strategy [3], since our objective was to collect early evidence and practitioner feedback on the usefulness of *PSC* to assess the propensity to generate code smells at the snippet level. The participants were selected based on their combined experience in machine learning and software engineering. We prioritized relevant expertise over randomness to ensure a meaningful interpretation of the metric output. Invitations were sent by email across academic and professional networks, and recipients were encouraged to share the survey with peers.

4.5.3 *Data Collection.*

We contacted approximately 110 potential participants from academic and industry settings, in order to collect perspectives from individuals with varying levels of experience in software engineering. Invitations were sent by email and participants were not informed of the specific goals of the study during recruitment. The survey was administered using Qualtrics. In total, we received 49 responses: 25 for the control survey and 24 for the treatment survey. After filtering out incomplete or low-quality submissions, such as those with unanswered questions or only demographic information, we retained 36 valid responses. The final dataset consisted of 18 control responses and 18 treatment responses.

4.5.4 *Survey Structure.*

The survey consisted of three sections: background profiling, code evaluation, and post-task reflection. Both the control and treatment groups followed the same structure, with the only distinction being the inclusion of the *PSC* score and two additional questions in the treatment. The survey began by collecting background information such as the current role of the participants, years of programming experience, familiarity with Python and code smells, and prior use of LLMs in software development workflows. In the second section, five Python code snippets were shown to the participants, each associated with a specific code smell detected using `Pylint`. The control group viewed the code along with a textual description of the smell, while the treatment group also saw the corresponding *PSC* score. For each snippet, all participants answered three questions: (Q_1) *"How likely do you think this code smell was introduced systematically or intentionally by the model (rather than being accidental)?"*; (Q_2) *"How important do you think it is to review or fix this code smell in a real-world codebase, considering the potential risk of similar issues reappearing in the future?"*; and (Q_3) *"How confident are you in your judgment about this code smell?"* The treatment group received two additional questions: (Q_4) *"Did the PSC score influence your judgment about this code smell?"* and (Q_5) *"Please briefly explain how the PSC score influenced

Table 1: *LLMs* used in experiments.

Same Size			*Same Architecture*		
ID	**Model**	**Size**	**ID**	**Model**	**Size**
M_1	CodeLlama-7b-hf [31]	7B	S_1	Qwen2.5-Coder-0.5B [27]	0.5B
M_2	Mistral-7B-v0.3 [10]	7B	S_2	Qwen2.5-Coder-1.5B [27]	1.5B
M_3	Qwen2.5-Coder-7B [27]	7B	S_3	Qwen2.5-Coder-3B [27]	3B
M_4	starcoder2-7b [17]	7B	S_4	Qwen2.5-Coder-7B [27]	7B

Table 2: List of *SECT*

Transformation	Description	Original	Transformed
Add2Equal	Convert add/subtract assignments to equal assignments	`a += 9`	`a = a + 9`
SwitchEqualExp	Switch the two expressions on both sides of the infix expression whose operator is ==	`a == b`	`b == a`

InfixDividing	Divide an in/pre/post-fix expression into two expressions	`x = a + b * c`	`temp = b *c`
			`x = a + temp`

SwitchRelation	Transform relational expressions	`a >b`	`b <a`
RenameVariable-1	Replace a variable name by its first character	`number = 1`	`n = 1`
RenameVariable-2	Replace a variable name by substitutions from CodeBERT	`number = 1`	`myNumber = 1`

your decision." The final section of the survey asked participants to reflect on their general trust in *LLM*-generated code and their interest in integrating automated smell detection tools into development workflows.

4.5.5 *Statistical Analysis.*

To evaluate the impact of *PSC* exposure on participant responses, we applied non-parametric statistical tests suitable for ordinal data. Specifically, we used the Mann–Whitney U test to compare responses between the control and treatment groups for each of the three main questions (Q_1–Q_3). This test does not assume normality and is appropriate for detecting differences in central tendency between independent samples with Likert scale results. Statistical comparisons were performed independently for each question and code smell type, yielding a total of 15 comparisons (3 questions in 5 snippets). For each comparison, we report the p value and highlight statistically significant differences ($p < .05$). Open-ended responses from Q_5 were analyzed using thematic analysis [6], with two authors independently coding answers and consolidating recurring ideas into broader themes.

4.5.6 *Survey Validity.*

We validated the survey through a pilot study with four doctoral students experienced in SE and familiar with machine learning models. None of them were involved in this work. They completed both control and treatment versions and provided feedback on question clarity, code readability and the interpretability of the *PSC* explanation. Their input led to revisions in prompt phrasing, improvements to the survey layout and refinements to the wording used to describe *PSC*.

5 Experimental Context

This section outlines the empirical setting for our study and details the data, models, and experimental conditions used to address our research questions. We begin by describing the construction of the data set used for analysis. We then present the models and treatment interventions evaluated throughout the quantification, explanation, mitigation, and usability tasks.

5.1 Dataset

Our experiments are based on an expanded version of the *CodeSmellData* corpus [37], referred to as *CodeSmellData 2.0*. This dataset was constructed through a multistage pipeline designed to extract and annotate large-scale Python code snippets from open-source repositories, allowing empirical evaluation of code smell generation in *LLM*-generated code.

Data Mining and Processing. We began by extracting method-level Python snippets from public GitHub repositories with more than 1K stars, published between January 2020 and January 2025. This process yielded approximately 4.7M methods, which were stored in a MySQL database. Each snippet was processed to compute syntactic and lexical confounders using the GALERAS schema [30]. In addition to these features, we extracted part-of-speech (POS) token counts using the `spaCy` parser to support fine-grained structural and linguistic analysis.

Deduplication and Annotation. We removed exact and semantically similar duplicates by comparing abstract syntax tree (AST) representations, resulting in a deduplicated set of approximately 2.1M unique methods exported to JSON format. Methods that did not perform the AST parsing were excluded. The remaining snippets were analyzed using `Pylint` to identify code smells, recording their types and sources. This step produced around 2.8M unique code smell instances, which collectively comprise *CodeSmellData 2.0* [14].

Filtering for Experimental Use. Due to computational constraints and the need for uniform sampling, we applied two additional filters to produce the dataset used in our experiments. First, we excluded any snippet that exceeded 700 tokens when tokenized using the `CodeLlama-7b-hf` (M_1) tokenizer. Second, for each type of code smell, we retained exactly 500 randomly sampled instances. Smell types with fewer than 500 examples were excluded from the analysis. The resulting evaluation dataset includes 41 distinct smell types covering the **warning**, **convention**, and **refactor** categories defined by `Pylint`.

Table 3 lists the included smell types and their `Pylint` identifiers. Each type is capped at 500 instances for consistency. However, the full distribution of smells from *CodeSmellData 2.0* is preserved in the dataset released [14].

5.2 Models

We selected pre-trained decoder-only models that use causal autoregressive decoding, making them well suited for the token-level probability computations required by *PSC*. These architectures are standard in code generation due to their ability to model long-range dependencies. The chosen models support controlled comparisons across architecture and scale. Table 1 summarizes the full set of selected models.

Although our implementation of *PSC* is designed for autoregressive decoding with access to token probabilities, the method is not limited to this setting. The metric can be extended to other types of generative architecture, including encoder-only or encoder-decoder models, provided that token-level likelihoods are accessible during inference.

5.3 Causal Settings

Each intervention in T (refer to Fig. 2) is designed to evaluate the influence of specific factors that we hypothesize contribute to *PSC*. These factors include: T_1, the type of generation; T_2, the model size;

T_3, the model architecture (depicted in Fig. 3); and T_4, the type of prompt (depicted in Fig. 4). We define the interventions for each treatment as follows.

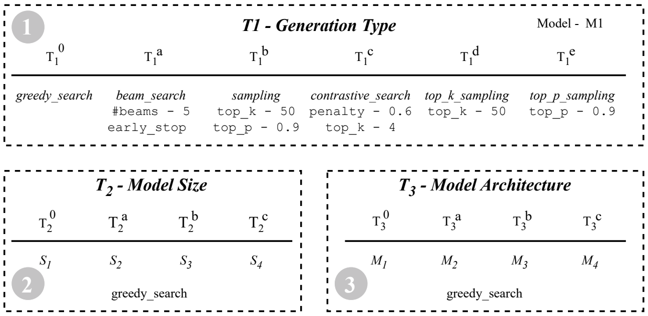


Figure 3: Causal Analysis T_{1-3} Settings

5.3.1 Intervention Setting $P(Y_1|do(T_1)$. The T_1 intervention (see ① in Fig. 3) manipulates the generation strategy used by the language model during inference. We evaluated six decoding methods: *greedy search*, *beam search*, *sampling*, *contrastive search*, *top-k sampling*, and *top-p sampling*. The beam search is configured with five beams and early stopping enabled. Contrastive search uses a penalty of 0.6 and a top-k value of 4. The top$k$ sampling is applied with $k = 50$, and the top$p$ sampling uses $p = 0.9$. All decoding strategies are applied to the same model instance (M_1) to ensure comparability between conditions.

For the code snippet in the dataset (Sec. 5.1), we truncate 50% of its original token length and instruct the model to complete the remaining portion using a specific decoding strategy. This 50% cutoff provides a consistent balance between context and generation length across examples. We then apply `Pylint` to the completed full snippet to detect any code smells. Although it is possible that the prefix may already contain smells, we analyze only those that appear in the generated portion. Smells are retained for analysis only if their source location falls entirely within the model-generated segment. For each of these, we compute the corresponding *PSC* ($Y_1 = f(\theta_\mu^{\text{median}})$) to quantify the propensity of the model to produce the detected issue.

We define *greedy search* as the control condition, denoted by T_1^0. Each alternative decoding strategy serves as a separate treatment condition, denoted T_1^a through T_1^e. We estimate average treatment effects (*ATEs*) by comparing each T_1^{a-e} setting against the control.

5.3.2 Intervention Setting $P(Y_1 \mid do(T_2))$. The T_2 intervention (see ② in Fig. 3) examines the effect of model size on code smell propensity. We evaluated four variants of the `Qwen2.5-Coder` model family, ranging from $0.5B$ to $7B$ parameters. These are denoted S_1 through S_4 in Table 1. To isolate the effect of model size, all models share the same architecture and are evaluated under a fixed decoding strategy (*greedy search*).

For each model, we compute the *PSC* ($Y_1 = f(\theta_\mu^{\text{median}})$) for every code snippet that contains a smell instance in the evaluation dataset, treating the snippet as a fixed input rather than generating it from scratch. Using an identical code across models controls the variability of generation and isolates the causal effect of the size of the model on *PSC*.

In the causal setting, the smallest model S_1 serves as the control condition, denoted T_2^0. The remaining models S_2, S_3, and S_4 are treated as treatment conditions, labeled T_2^a, T_2^b, and T_2^c, respectively. We estimate average treatment effects by comparing the distribution of Y_1 across these size-based configurations.

5.3.3 Intervention Setting $P(Y_1 \mid do(T_3))$. The T_3 intervention (see ③ in Fig. 3) explores the influence of model architecture on code smell propensity. We evaluated four language models with distinct architectural designs, denoted M_1 through M_4, and selected to be of comparable size (approximately 7B parameters). To control for generation variability, all models are evaluated using the same decoding strategy (*greedy search*).

Following the same rationale as in the T_2 intervention, we treat each code fragment in the evaluation data set as a fixed input and compute the *PSC* ($Y_1 = f(\theta_\mu^{\text{median}})$). This allows for a fair comparison of architectures under identical input conditions.

In the causal setting, model M_1 serves as the control condition, denoted T_3^0. The remaining models (M_2, M_3, and M_4 are treated as intervention conditions, labeled T_3^a, T_3^b, and T_3^c, respectively. We estimate the average treatment effect by comparing the distribution of Y_1 across these architectural variants.

5.3.4 Intervention Setting $P(Y_1 \mid do(T_4))$. The T_4 intervention (see ④ in Fig. 4) investigates the influence of prompt type on code smell propensity. We define four prompt formulations that vary in their level of instruction and emphasis on code quality. The simplest version presents only a function placeholder (`<smelly_function>`) with no additional context. The second variant appends a generic instruction: *'Complete the following code'*. The third introduces a role-based preamble indicating that the user is an expert software engineer committed to producing clean, maintainable, and production-quality code. The fourth and most structured prompt explicitly lists code smells to avoid and includes best practices for modular and readable Python.

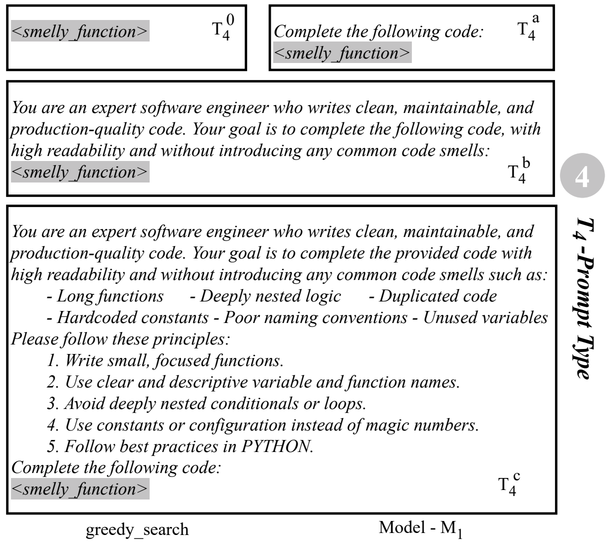
Similarly to interventions for T_2 and T_3, we isolate the effect of prompt design on *PSC* by evaluating all prompt variants using the same model (M_1) and a fixed decoding strategy (*greedy search*). For each prompt condition, we compute the *PSC* for every snippet in the evaluation dataset, treating the snippet as the model output given the corresponding prompt.

In causal analysis, the minimal prompt (`<smelly_function>`) is treated as the control condition, denoted T_4^0. The remaining three prompts are treated as intervention conditions: T_4^a, T_4^b, and T_4^c, reflecting increasing degrees of instructional guidance. We estimate the average treatment effects by comparing the resulting Y_1 distributions across prompt types.

6 Results & Discussion

Table 3 summarizes the results of our ANOVA analysis, which evaluates the robustness of the *PSC* metric under *SECT*. For 31 out of 41 code smell types (76%), we observe p-values of 1 and effect sizes η^2 close to zero. This indicates that the distribution of the *PSC* scores remains consistent between syntactic variations that preserve the semantics of the code. These findings suggest that *PSC* captures stable model behavior and reflects deeper generative tendencies rather than shallow token patterns. Such stability is

Figure 4: Causal Analysis, T_4 Settings



<smelly_function>

Complete the following code:
<smelly_function>

You are an expert software engineer who writes clean, maintainable, and production-quality code. Your goal is to complete the following code, with high readability and without introducing any common code smells:
<smelly_function>

You are an expert software engineer who writes clean, maintainable, and production-quality code. Your goal is to complete the provided code with high readability and without introducing any common code smells such as:
- Long functions - Deeply nested logic - Duplicated code
- Hardcoded constants - Poor naming conventions - Unused variables

Please follow these principles:
1. Write small, focused functions.
2. Use clear and descriptive variable and function names.
3. Avoid deeply nested conditionals or loops.
4. Use constants or configuration instead of magic numbers.
5. Follow best practices in PYTHON.

Complete the following code:
<smelly_function>

Table 3: SEPT robustness F_{anova}test results (*i.e.,* RQ_1).

ID	F_{anova}	p_{value}	η^2	$CI_{95\%}$
C0116	0	1	0	0.04 ± 0.003
R0917	0	1	0	0.13 ± 0.007
R1710	0	1	0	0.03 ± 0.003
R0914	0	1	0	0.03 ± 0.003
W0102	0	1	0	0.03 ± 0.003
C0114	0	1	0	0.24 ± 0.009
R0912	0	1	0	0.03 ± 0.003
R0913	0	1	0	0.12 ± 0.007
W0104	0	1	0	0.52 ± 0.017
R1735	1.98e-3	1	0	0.12 ± 0.007
W0622	2.72e-3	1	0	0.19 ± 0.011
C0415	7.34e-3	1	0	0.58 ± 0.012
C0303	8.72e-3	1	0	0.04 ± 0.006
W0601	1.02e-2	1	0	0.35 ± 0.011
C0305	1.7e-2	1	3e-5	0.87 ± 0.009
C2801	1.95e-2	1	3e-5	0.81 ± 0.011
C0304	2.44e-2	1	1.2e-3	0.51 ± 0.01
W0718	2.72e-2	1	4e-5	0.67 ± 0.012
W1406	3.61e-2	1	7e-5	0.74 ± 0.011
W0611	4.08e-2	1	7e-5	0.66 ± 0.012
W0621	5.41e-2	1	1.2e-4	0.39 ± 0.013
C3001	6.18e-2	1	1.2e-4	0.79 ± 0.009
W1309	6.39e-2	1	1.3e-4	0.66 ± 0.012
W0707	1.03e-1	9.81e-1	1.9e-4	0.81 ± 0.009
W0612	1.25e-1	9.74e-1	2.8e-4	0.43 ± 0.014
R1732	1.66e-1	9.56e-1	3.2e-4	0.77 ± 0.011
W0311	2.76e-1	8.94e-1	5.2e-4	0.72 ± 0.011
W0613	2.82e-1	8.9e-1	4.5e-4	0.27 ± 0.011
C0301	0.36	0.83	6.3e-4	0.76 ± 0.011
W0511	0.43	0.78	8.0e-4	0.37 ± 0.014
C0200	0.61	0.65	1.0e-3	0.96 ± 0.003
C0325	0.89	0.47	1.64e-3	0.75 ± 0.010
W0719	2.59	0.03	4.52e-3	0.72 ± 0.010
C0123	3.29	1.5e-2	0.13	0.71 ± 0.013
R1720	3.75	5e-3	6.35e-3	0.87 ± 0.005
R1705	3.93	3.49e-3	6.56e-3	0.88 ± 0.005
C0209	33.31	0	6.62e-2	0.88 ± 0.005
W0212	55.94	0	0.1	0.79 ± 0.011
C0321	65.85	0	0.12	0.72 ± 0.012
W1514	68.86	0	0.13	0.77 ± 0.012
C0103	94.51	0	0.18	0.32 ± 0.013

background grey: reduced *PSC* robustness

essential for a quality metric intended to evaluate the structural properties of code rather than incidental formatting.

However, a subset of smell types exhibits notable sensitivity. Specifically, `C0103` (*invalid*), `C0209` (*considering*), `W0212` (*protected*), and `C0321` (*multiple*) show higher effect sizes ($\eta^2 \geq 0.1$) and p-values below .05, with confidence intervals that vary more widely than in stable smells. These results, highlighted in gray in Table 3, indicate reduced robustness. In these cases, minor syntactic changes shift the model's token likelihoods enough to affect the resulting *PSC* scores. Many of these smells depend on naming, formatting, or other localized syntactic patterns, which explains the increased variability in their confidence intervals. This pattern reflects a broader distinction between *semantic* and *syntactic* smells: semantic smells tend to remain stable under surface edits, whereas syntactic smells rely more directly on lexical and formatting characteristics and are therefore more sensitive to *SECT*.

Across all types of smell, the average *std* of the *PSC* scores was highest for *SwitchRelation*, *Add2Equal*, *SwitchEqualExp*, and *InfixDividing*, each averaging around 0.65. In contrast, variability was lower for renaming-based transformations, with *RenameVariable-1* averaging 0.44 and *RenameVariable-2* averaging 0.59.

To assess the explanatory value of *PSC* relative to existing metrics, we computed the information gain (*IG*) between each score (PSC, BLEU and CodeBLEU) and a binary label representing the severity of the smell of the code (see Sec. 4.2). The severity label assigns a positive class to snippets where more than 50 percent of the tokens are marked as smelly. Fig. 5 presents the results for all types of smells.

We find that *PSC* consistently produces a higher information gain than BLEU and CodeBLEU. This suggests that *PSC* is more effective in explaining the presence of severe code smells, offering a stronger and more discriminative signal about quality. Although BLEU and CodeBLEU focus on surface similarity with a reference implementation, they are less responsive to structural concerns such as maintainability or design violations. In contrast, *PSC* reflects the model's internal confidence in generating smelly tokens, making it more directly aligned with latent quality dimensions.

The results indicate that *PSC* offers two desirable properties for evaluating generated code: **robustness** to superficial edits and **alignment** with structural quality indicators. *PSC* demonstrates strong reliability under semantic-preserving transformations and provides more informative signals than reference-based metrics. Its sensitivity to certain smell types highlights the impact of local syntax on token prediction, but overall it offers a more interpretable and quality-aligned alternative for evaluating generated code.

> ***RQ*$_1$ [Measure]**: *PSC* provides a reliable way to measure the likelihood of code smell generation in *LLM* outputs. *PSC* remains robust under syntactic variation for 76% of smell types and yields higher information gain than BLEU and CodeBLEU, indicating stronger alignment with smell severity and code quality.

Table 4 reports the average treatment effects (*ATEs*) estimated across four causal interventions (Sec. 5.3). Results indicate varying sensitivity across code smell types, with some interventions producing stronger effects than others.

Generation type (T_1) leads to substantial variation in *PSC* scores depending on the decoding strategy. Smells such as `C2801` (*unnecessary -dunder-call*) and `W1406` (*redundant-u-string-prefix*) show lower *PSC* values when sampling-based methods like T_1^c or T_1^e are used, suggesting reduced smell propensity under stochastic generation. Conversely, formatting-related smells, including `C0303` (*trailing-whitespace*) and `C0304` (*missing-final-newline*), tend to increase under non-greedy decoding, reflecting greater variability in surface-level code structure.

Model size (T_2) exhibits relatively weak influence on the *PSC*. Most treatment effects remain close to zero, and even slight improvements observed for convention-type smells lack statistical significance. These findings suggest that scaling the number of parameters alone does not substantially reduce the likelihood of generating smelly code, at least within the model sizes tested. The assumption that larger models automatically produce cleaner outputs is not supported.

Figure 5: Information Gain (*IG*) Results (*i.e.,* RQ$_1$).

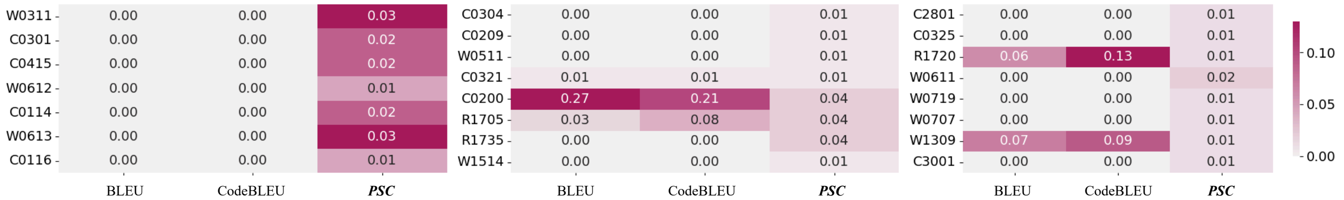


Table 4: Causal Effects (*ATEs*) (*i.e.,* RQ$_2$)

Generation Type

	T_1^a		T_1^b		T_1^c		T_1^d		T_1^e	
ID	ρ	***ATE***	ρ	***ATE***	ρ	***ATE***	ρ	***ATE***	ρ	***ATE***
C0303	**.38**	.15	**.44**	.21	**.51**	.23	**.52**	.28	**.51**	.26
C0304	**.27**	.1.0	**.55**	.30	**.32**	.17	**.24**	.15	**.31**	.17
C2801	-.03	0.0	**.12**	-.01	.04	-.14	.02	-.04	.07	-.10
R1732	**-.11**	0.0	-.01	-.06	-.09	-.12	-.07	-.06	-.04	.05
R1735	-.04	-.03	**.10**	-.03	0.0	-.08	-.01	-.10	.03	.09
W0104	**.10**	.1.0	**.33**	.06	**.16**	.04	.09	.01	**.22**	.13
W0511	-.07	-.07	**.35**	.24	.02	-.07	.04	0.0	.03	-.04
W0612	.03	.12	**.25**	.13	**.19**	.21	**.13**	.19	**.20**	.19
W0613	.08	.05	**.13**	-.01	**.25**	.16	**.16**	.19	**.30**	.19
W0621	-.05	0.0	**.12**	.10	.07	.12	.07	.14	.07	.12
W1309	**-.35**	-.05	.07	-.10	.02	-.02	-.09	-.12	0.0	-.17
W1406	**.20**	.15	**.20**	0.0	**.12**	.03	**.13**	.40	**.16**	-.16

Model Size

	T_2^a		T_2^b		T_2^b	
ID	ρ	***ATE***	ρ	***ATE***	ρ	***ATE***
C0103	.04	.10	.06	.17	.08	-.02
C0200	**.20**	.10	**.25**	.11	**.34**	.14
C0209	.08	.05	**.10**	.12	**.16**	.15
C0301	**.13**	.12	**.16**	.25	**.23**	.30
C0304	.06	.01	.08	.16	**.14**	.33
C0321	.09	.28	**.11**	.39	**.16**	.42
C0325	**.10**	.04	**.13**	.13	**.18**	.16
W0511	.05	.07	.06	.12	.09	.14
W0613	.06	.21	**.13**	.33	**.10**	.31
W1514	.05	.11	.04	.10	.08	.13

Model Architecture

	T_3^a		T_3^b		T_3^c	
ID	ρ	***ATE***	ρ	***ATE***	ρ	***ATE***
C0103	**-.10**	.24	**.10**	.04	**.17**	32
C0114	**-.23**	-.14	**-.31**	-.14	.08	.06
C0304	**-.14**	-.53	**-.15**	-.25	**-.22**	-.55
C0305	-.07	-.03	-.04	-.02	**-.40**	-.26
C0321	-.06	.06	**-.11**	.15	-.02	.18
C0415	**-.18**	-.16	**-.16**	-.38	**-.18**	-.33
C3001	**-.15**	-.19	**-.13**	-.24	**-.10**	-.07
R0917	**-.27**	-.19	.06	.08	**.30**	.30
R1705	**-.18**	-.05	**-.35**	-.17	**-.25**	-.15
R1720	**-.15**	-.03	**-.35**	-.16	**-.28**	-.11
W0601	**-.20**	-.23	**-.15**	-.14	**-1.0**	-.23
W0612	**-.20**	-.22	**-.15**	-.14	**-1.0**	-.07
W0707	**-.12**	-.17	**-.29**	-.39	**-.32**	-.34
W1406	-.06	-.09	**-.18**	-.27	**-.11**	-.14
W0719	-.09	-.17	**-.26**	-.10	**-.19**	-.06

Prompt Type

	T_4^a		T_4^b		T_4^c	
ID	ρ	***ATE***	ρ	***ATE***	ρ	***ATE***
C0209	-.01	-.05	-.03	-.06	-.03	-.10
C0304	-.07	-.50	**-.10**	-.52	**-.10**	-.52
C0321	0.0	.13	-.01	.15	-.01	.10
C0325	-.02	-.01	-.04	-.12	-.04	-.11
R0917	-.05	-.04	-.06	-.05	**-.18**	-.13
W0102	.08	.06	.09	.06	**.22**	.19
W0601	-.08	-.06	**-.12**	-.06	**-.10**	.11
W0611	-.04	-.12	-.08	-.12	-.07	-.11
W0707	0.0	-.01	-.02	-.13	-.01	-.12
W0719	-.02	-.05	-.01	-.13	-.02	-.15

bold: – correlation, **bold underlined**: + correlation, background purple: – causal effect, background grey: + causal effect

Model architecture (T_3) has a more pronounced impact. Several refactor and warning-related smells, including `R1705` (*no-else-return*), `W0612` (*unused-variable*), and `W0707` (*raise-missing-from*), display large negative treatment effects across architectural variants. Changes in network design influence the model's internal token distributions, affecting structural and semantic decisions during code synthesis. Even stylistic issues, such as `C0304` and `C0325`, are affected, reinforcing the role of architecture in shaping code quality.

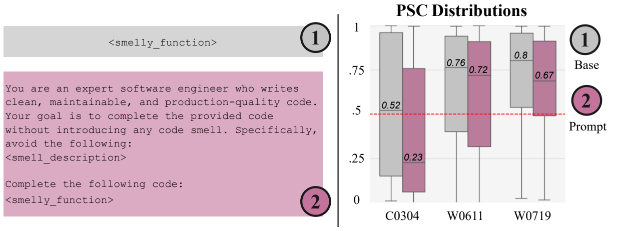
Prompt design (T_4) consistently reduces smell propensity, particularly when instructions emphasize best practices and explicitly discourage problematic patterns. Strongest effects are observed for `W0612`, `W0707`, and `C0321`, where more structured prompts yield noticeable reductions in *PSC* scores. Prompt variant T_4^c, which includes specific smell avoidance guidance, produces the most substantial improvement. Instructional framing thus emerges as an effective lever for steering generation without retraining the model.

Model architecture and prompt formulation emerge as effective levers for mitigating code smell generation, while model size offers minimal benefit. Decoding strategy shows mixed effects, with improvements for some semantic issues and degradation for surface-level ones. Prompt engineering, in particular, offers a practical intervention for developers, requiring no changes to the model itself. Architectural findings suggest deeper interactions between pretraining design and output quality, opening avenues for more architecture-aware alignment strategies.

Refutation tests confirm the stability of the estimated *ATEs*. Results remain consistent across validation checks, indicating robustness to noise and unmeasured factors.

> ***RQ***$_2$ **[Explain]**: The *ATEs* reveal that the *PSC* is differentially influenced by generation type, model size, architecture, and prompt design. Although model size shows limited impact, architectural changes and prompt formulation exhibit strong and consistent effects on reducing the propensity for specific code smells.

Figure 6: Mitigation Case Study Results (*i.e.,* RQ$_3$).



```
<smelly_function>
```

```
You are an expert software engineer who writes
clean, maintainable, and production-quality code.
Your goal is to complete the provided code
without introducing any code smell. Specifically,
avoid the following:
<smell_description>

Complete the following code:
<smelly_function>
```

Fig. 6 shows the impact of prompt-based mitigation on the *PSC* score for three smell types: `W0719` (*broad-exception-raised*), `C0304` (*missing-final-newline*) and `W0611` (*unused-import*). The prompt instructs the model to complete the provided code without introducing any of the listed smells. To assess its effect, we constructed a new evaluation set of 500 examples for each smell type. All generations were produced using model M_1 with greedy decoding under two conditions: a baseline prompt containing only the original function (①) and a mitigation prompt including the instruction (②).

The boxplots in Fig. 6 compare the resulting *PSC* distributions across both conditions. For `W0719`, the median *PSC* decreases from .80 under the baseline to .67 with mitigation. For `C0304`, the median decreases from .76 to .72, and for `W0611` the reduction is more pronounced, from .52 to .23. In all three cases, the distributions shift downward, with a larger proportion of instances falling below the smelly threshold $\lambda = 0.5$.

Table 5: User Study Results (*i.e.,* RQ$_4$).

ID	Q_{ID}	*U-stat*	p_{value}
C0209	1	138	.437
C0209	2	154	.726
C0209	3	136	.393
W0613	1	162	1
W0613	2	84.5	.034
W0613	3	67.5	.002
W0612	1	89	.002
W0612	2	113.5	.422
W0612	3	153	.771
W0102	1	186.5	.429
W0102	2	130	.863
W0102	3	172	.751
W0622	1	230	.026
W0622	2	78.5	.021
W0622	3	167	.881

background purple: – statistically significant

Because this intervention relies exclusively on prompt design, it requires no additional training cost and applies directly at inference time. The consistent declines across the three smell types suggest that prompt-based mitigation is an effective and scalable strategy for improving the quality of *LLM*-generated code.

> ***RQ*$_3$ [Mitigate]**: Prompt-based mitigation reduces the median *PSC* for `W0719` from .80 to .67, for `C0304` from .76 to .72, and for `W0611` from .52 to .23. These results show that carefully crafted prompts reduce the model's propensity to generate code smells during inference.

We assessed whether the *PSC* score influences how developers evaluate code smells through a between-subjects user study comparing two conditions: a control group that viewed code samples without the *PSC* score, and a treatment group that received the same samples with the *PSC* score displayed. Table 5 summarizes the statistical results, with significant p-values ($p < .05$) highlighted in purple.

Five significant effects were observed across three smell types. For `W0613` (*unused-argument*), access to the *PSC* score increased the perceived importance of fixing the smell (Q_2, $p = .034$) and boosted participants' confidence in their judgment (Q_3, $p = .002$). For `W0612` (*unused-variable*), participants in the treatment group were more likely to attribute the smell to systematic model behavior (Q_1, $p = .002$). In the case of `W0622` (*redefined-builtin*), participants exposed to the score more frequently judged the smell as systematically introduced (Q_1, $p = .026$) and also rated it as more important to address (Q_2, $p = .021$).

Responses to Q_4 suggest that the *PSC* score helped participants reassess their initial judgments, especially when the smell was subtle or unfamiliar. Some used it as a reference point to confirm concerns or adjust prioritization, with high scores increasing urgency and low scores offering reassurance. Overall, the score functioned as a heuristic for resolving uncertainty in edge cases.

The *PSC* score influenced how developers interpreted and prioritized code smells, especially for subtle issues like variable usage and naming. Its impact varied by smell type, suggesting that *PSC* is most useful in ambiguous cases rather than as a universal signal.

> ***RQ*$_4$ [Usefulness]**: Exposure to the *PSC* helped participants identify smells as model-introduced, prioritize them for resolution, and evaluate them with greater confidence. The perceived usefulness of *PSC* varied by smell type.

7 Threats to Validity

Threats to **internal validity** concern the correctness of our implementation, causal reasoning, and study design. To mitigate these, we leverage established tools such as `doWhy` for causal inference and carefully validate our *PSC* implementation. We perform multiple robustness checks, including placebo tests, randomized confounding, and subset validation, to ensure the reliability of our causal estimates. In the user study, we controlled for potential ordering effects and standardized question phrasing to minimize participant bias.

External validity relates to the extent to which our findings generalize beyond the specific settings of our study. Although our experiments focus on Python and decoder-only language models, we incorporate diverse architectures and analyze 41 distinct Pylint smell types (Table 3), which enhances representativeness within our chosen domain. Generalization to other programming languages or tooling ecosystems remains an avenue for future work.

Construct validity addresses how accurately our metrics capture the underlying concept of code quality. We quantify smell propensity through token-level smell density and estimate generation likelihood using *PSC*, grounded in static analysis techniques. The alignment between these metrics and perceived quality is supported by robustness analysis and user study feedback. While *PSC* builds on established definitions of code smells and reflects structural indicators of quality, our causal analysis depends on a confounder set drawn from features most commonly associated with smell occurrence. Other design attributes such as member hierarchy, graph call patterns, or maintainability indices were not included because they cannot be reliably extracted from isolated snippets. These omissions may introduce residual confounding, although refutation tests suggest that our estimates remain stable. Future work could incorporate richer design-level signals to broaden the confounder set.

8 Conclusions & Future Work

In this paper, we used *PSC* to measure, explain, and mitigate the generation of code smells in *LLMs*. Our findings show that *PSC* aligns more closely with structural quality than surface-level metrics such as BLEU and CodeBLEU, and remains robust under semantic-preserving transformations. Causal analysis revealed that prompt design and model architecture are key factors that influence the propensity to smell, while the size of the model has limited impact. We also demonstrated that prompt-based mitigation can significantly reduce the likelihood of generating smelly code. A user study confirmed that *PSC* improves developer judgement in ambiguous scenarios.

Future work includes extending *PSC* to other programming languages and model architectures, which will require adapting smell taxonomies and ensuring access to next-token probabilities. Another direction is the development of full mitigation plans that move beyond the case study explored in this work. More broadly, *PSC* has potential as a lightweight interpretability signal beyond code quality, offering a practical way to examine how model probability distributions behave and what these patterns reveal about underlying generation tendencies.

References

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. arXiv:2207.04237 [cs.SE] https://arxiv.org/abs/2207.04237

[2] Ahmed Aljohani and Hyunsook Do. 2024. From Fine-tuning to Output: An Empirical Investigation of Test Smells in Transformer-Based Test Code Generation. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) *(SAC '24)*. Association for Computing Machinery, New York, NY, USA, 1282–1291. doi:10.1145/3605098.3636058

[3] Sebastian Baltes and Paul Ralph. 2021. Sampling in Software Engineering Research: A Critical Review and Guidelines. arXiv:2002.07764 [cs.SE] https://arxiv.org/abs/2002.07764

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[5] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4818–4837. doi:10.1109/TSE.2021.3128234

[6] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284. doi:10.1109/ESEM.2011.36

[7] Benedetta Donato, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2025. Studying How Configurations Impact Code Generation in LLMs: the Case of ChatGPT. arXiv:2502.17450 [cs.SE] https://arxiv.org/abs/2502.17450

[8] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software* 203 (Sept. 2023), 111741. doi:10.1016/j.jss.2023.111741

[9] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[10] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[11] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. arXiv:2303.07263 [cs.SE] https://arxiv.org/abs/2303.07263

[12] Mohammed Kharma, Soohyeon Choi, Mohammed AlKhanafseh, and David Mohaisen. 2025. Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. arXiv:2502.01853 [cs.CR] https://arxiv.org/abs/2502.01853

[13] Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. 2024. The Impact of Prompt Programming on Function-Level Code Generation. arXiv:2412.20545 [cs.SE] https://arxiv.org/abs/2412.20545

[14] SEMERU Lab. 2024. CodeSmellExt. https://github.com/WM-SEMERU/CodeSmellExt. Accessed: 2025-02-12.

[15] Thanh Le-Cong, Dat Nguyen, Bach Le, and Toby Murray. 2024. Towards Reliable Evaluation of Neural Program Repair with Natural Robustness Testing. arXiv:2402.11892 [cs.SE] https://arxiv.org/abs/2402.11892

[16] Feng Lin, Dong Jae Kim, Tse-Husn, and Chen. 2024. SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents. arXiv:2403.15852 [cs.SE] https://arxiv.org/abs/2403.15852

[17] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[18] Eric L. Melin, Adam J. Torek, Nasir U. Eisty, and Casey Kennington. 2024. Precision or Peril: Evaluating Code Quality from Quantized Large Language Models. arXiv:2411.10656 [cs.SE] https://arxiv.org/abs/2411.10656

[19] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H. Sarker, Leandros Maglaras, and Naeem Janjua. 2024. Can We Trust Large Language Models Generated Code? A Framework for In-Context Learning, Security Patterns, and Code Evaluations Across Diverse LLMs. arXiv:2406.12513 [cs.CR] https://arxiv.org/abs/2406.12513

[20] Henrique Nunes, Eduardo Figueiredo, Larissa Rocha, Sarah Nadi, Fischer Ferreira, and Geanderson Esteves. 2025. Evaluating the Effectiveness of LLMs in Fixing Maintainability Issues in Real-World Projects. arXiv:2502.02368 [cs.SE] https://arxiv.org/abs/2502.02368

[21] Wendkûuni C. Ouédraogo, Yinghua Li, Kader Kaboré, Xunzhu Tang, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2024. Test smells in LLM-Generated Unit Tests. arXiv:2410.10628 [cs.SE] https://arxiv.org/abs/2410.10628

[22] David N. Palacio, Daniel Rodriguez-Cardenas, Alejandro Velasco, Dipin Khati, Kevin Moran, and Denys Poshyvanyk. 2024. Towards More Trustworthy and Interpretable LLMs for Code through Syntax-Grounded Explanations. arXiv:2407.08983 [cs.SE] https://arxiv.org/abs/2407.08983

[23] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. [Journal First] On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 482–482. doi:10.1145/3180155.3182532

[24] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. arXiv:2108.09293 [cs.CR] https://arxiv.org/abs/2108.09293

[26] Judea Pearl and Dana Mackenzie. 2018. *The Book of Why: The New Science of Cause and Effect* (1st ed.). Basic Books, Inc., USA.

[27] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[28] Leonardo Criollo Ramírez, Xavier Limón, Ángel J. Sánchez-García, and Juan Carlos Pérez-Arriaga. 2024. State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review. In *2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 331–339. doi:10.1109/CONISOFT63288.2024.00050

[29] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[30] Daniel Rodriguez-Cardenas, David N. Palacio, Dipin Khati, Henry Burke, and Denys Poshyvanyk. 2023. Benchmarking Causal Study to Interpret Large Language Models for Source Code. arXiv:2308.12415 [cs.SE] https://arxiv.org/abs/2308.12415

[31] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[32] Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim, Sourov Jajodia, and Joanna C. S. Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 71–82. doi:10.1109/SCAM55253.2022.00014

[33] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna C. S. Santos. 2024. Quality Assessment of ChatGPT Generated Code and their Use by

Developers. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 152–156.

[34] Wannita Takerngsaksiri, Micheal Fu, Chakkrit Tantithamthavorn, Jirat Pasuksmit, Kun Chen, and Ming Wu. 2025. Code Readability in the Age of Large Language Models: An Industrial Case Study from Atlassian. arXiv:2501.11264 [cs.SE] https://arxiv.org/abs/2501.11264

[35] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2024. Bugs in Large Language Models Generated Code: An Empirical Study. arXiv:2403.08937 [cs.SE] https://arxiv.org/abs/2403.08937

[36] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 403–414. doi:10.1109/ICSE.2015.59

[37] Alejandro Velasco, Daniel Rodriguez-Cardenas, Luftar Rahman Alif, David N. Palacio, and Denys Poshyvanyk. 2025. How Propense Are Large Language Models at Producing Code Smells? A Benchmarking Study. arXiv:2412.18989 [cs.SE] https://arxiv.org/abs/2412.18989

[38] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2021. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. arXiv:2009.06520 [cs.SE] https://arxiv.org/abs/2009.06520

[39] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM. doi:10.1145/3377811.3380429

[40] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 87–98. doi:10.1145/2970276.2970326

[41] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 334–345. doi:10.1109/MSR.2015.38

[42] 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. 1345–1357. doi:10.1145/3691620.3695508

[43] Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models. arXiv:2407.11470 [cs.SE] https://arxiv.org/abs/2407.11470

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009