

Investigating The Smells of LLM Generated Code^{*}

Debalina Ghosh Paul, Hong Zhu*, Ian Bayley

School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford, OX3 0BP, UK

Abstract

Context:

Large Language Models (LLMs) are increasingly being used to generate program code. Much research has been reported on the functional correctness of generated code, but there is far less on code quality.

Objectives:

In this study, we propose a scenario-based method of evaluating the quality of LLM-generated code to identify the weakest scenarios in which the quality of LLM generated code should be improved.

Methods:

The method measures code smells, an important indicator of code quality, and compares them with a baseline formed from reference solutions of professionally written code. The test dataset is divided into various subsets according to the topics of the code and complexity of the coding tasks to represent different scenarios of using LLMs for code generation. We will also present an automated test system for this purpose and report experiments with the Java programs generated in response to prompts given to four state-of-the-art LLMs: Gemini Pro, ChatGPT, Codex, and Falcon.

Results:

We find that LLM-generated code has a higher incidence of code smells compared to reference solutions. Falcon performed the least badly, with a smell increase of 42.28%, followed by Gemini Pro (62.07%), ChatGPT (65.05%) and finally Codex (84.97%). The average smell increase across all LLMs was 63.34%, comprising 73.35% for implementation smells and 21.42% for design smells. We also found that the increase in code smells is greater for more complex coding tasks and for more advanced topics, such as those involving object-orientated concepts.

Conclusion:

In terms of code smells, LLM's performances on various coding task complexities and topics are highly correlated to the quality of human written code in the corresponding scenarios. However, the quality of LLM generated code is noticeably poorer than human written code.

Keywords: Machine learning, Large language models, Performance evaluation, Code generation, Code quality, Usability, Code smell

1. Introduction

1.1. Motivation

Large language models (LLMs) are increasingly being used in practice to assist programmers with code generation. It is widely recognised that such machine learning (ML) models can significantly improve productivity [1], but there are concerns about the quality of the code generated. For example, the 2023 *Stack Overflow Developer Survey* conducted by Google in 2023 with over 90,000 respondents globally found that “39% of

developers say they don't trust AI-generated code” [2]. More recently, based on a quantitative study of 211 million lines of code changes submitted to GitHub between January 2020 and December 2024, Harding et al. [3] identified multiple “*signs of eroding code quality*” such as duplicated code and various forms of technical debt. Moreover, they were able to link this rising defect rate with AI adoption. So this raises an important question: what specific quality defects are present in LLM generated code?

Our previous work [4] has looked at two of these attributes: correctness and complexity, and evaluated ChatGPT on the benchmark ScenEval that we constructed, where correctness is determined by passing all test cases automatically generated from both ChatGPT generated code and the reference solution, and complexity was measured with cyclomatic complexity, cognitive complexity, and line counts. It was found that correctness was lower for more advanced coding topics and more complex coding tasks. ChatGPT-generated code was

^{*}This paper is an extended and revised version of the conference paper entitled “Does LLM Generated Code Smell?” to be published by IEEE in the *Proceeding of The 9th International Conference on Cloud, Big Data, and Communication Systems (ICCBDCS 2025)*. The results presented in the conference paper are preliminary and superseded by this paper.

^{*}Corresponding author

Email addresses: 19217422@brookes.ac.uk (Debalina Ghosh Paul), hzhu@brookes.ac.uk (Hong Zhu), ibayley@brookes.ac.uk (Ian Bayley)

more complex than human-written code. In addition, the complexity increases greater for more complex tasks than simpler ones. These suggest that LLMs are less useful for more complex and advanced tasks and that could explain why senior developers use them less often [5], [6].

However, Ziegler et al., in their study of GitHub Copilot, observed that the major driving force for the adoption of generated code is not its correctness but whether it is useful as a starting point for further development [7]. So, in this paper we will shift our attention to the quality attributes that are relevant to that. These include readability, testability, maintainability, ease of modification / evolution, reusability and so on.

1.2. Challenges And Our Approach

It is difficult to measure LLM generated code on these quality attributes since the context of their usage is unknown. Our solution is to use code smell detection techniques since they are well established in software engineering research for this purpose [8, 9, 10]. It is widely recognised that code smells are indicators of problems present in program code with maintenance, evolution and reuse [11].

A problem with the concept of code smells, however, is that it is relatively subjective. Beck and Fowler define that bad code smells are “*not precise criteria for flaws in program code*” [8, 12]. They suggest that the presence of smells is “*better to be judged based on informed human intuition*” and research has found that “*human agreement on smell detection is low*” [13]. Our solution is to follow best practice in ML research: benchmark LLM performance on a dataset and compare with a baseline. The dataset we will use is ScenEval [4]. It consists of tasks collected from both textbooks and questions submitted to StackOverflow; the latter is a professional coding problem-solving website so the questions are real-world. Each task is accompanied by a reference solution either written by the textbook authors or supplied by professional programmers in IT industry in response to the question on StackOverflow and scored highly by peers. These reference solutions provide a good baseline that reflects the current state-of-the-art in professionally written code so that we can decide whether the LLM-generated code is of comparable quality.

To achieve our research goal, it is insufficient to score each LLM with a single scalar value since they are used in many different contexts for different purposes by different users. So we evaluate the LLMs on different scenarios. These include different problem topics and different complexities. The information we need for filtering on these scenarios is included as metadata with each coding task in the ScenEval benchmark [4]. Different subsets of the dataset can therefore be formed easily to represent different scenarios.

However, the existence of these subsets necessitates repeated experiments, so we automate both the execution of the experiments and the subsequent analysis of the large volume of data that is produced. We design and implement an automated test system following the datamorphic testing methodology [14] and execute the experiment with the Morphy test automation environment [15].

1.3. Contributions

Our main contributions are as follows.

1. We propose a scenario-based method to investigate the quality of LLM generated code by detecting code smells, statistically analysing them and comparing them with a baseline of human-written programs. This method enables us to identify the quality weaknesses of LLM generated code specific to each scenario.
2. We have designed and implemented a test system to automate the experiments with LLMs and the analysis of the data obtained from the experiments.
3. We have conducted a systematic and intensive experiment with four current state-of-the-art LLMs and compared the generated code with human-written reference solutions in the benchmark as the baseline. The experiment demonstrates the validity and feasibility of the proposed method, and the efficiency and effectiveness of the test system.
4. We have identified the weaknesses of LLM for code generation with regards to the quality of the code generated. These results are the first of their kind in the literature as far as we know.

1.4. Structure of the Paper

The paper is organised as follows. Section 2 reviews related work on how to evaluate the quality of LLM-generated code and formulates the research questions. Section 3 gives a brief introduction to the notion of code smell and techniques for code smell detection. Our uses of these techniques are described. Section 4 presents the automated test system, which is designed and implemented based on the datamorphic software testing methodology. Section 5 presents the design of our experiment. Section 6 reports the results and presents the analysis of the data. Section 7 discusses the threats to experimental validity. Finally, Section 8 summarises the findings and discusses directions for future work.

2. Related Works And Open Problems

Evaluation of the capability of LLMs in code generation has hitherto focussed on functional correctness, but far less on code quality and only recently; see [16] for a recent review. We now discuss the few works in the literature that are relevant, followed by the open research questions addressed in this paper.

2.1. Manual Evaluation

In 2024, Miah and Zhu proposed a user-centric methodology to evaluate LLMs according to the quality of code generated [17]. The method consists of the following three components:

1. A *multi-attempt* testing process model: the tester engages in an iterative process of interactions with a LLM by (a) formulating, revising and submitting a query to the LLM under test, (b) getting responses from the LLM, (c) assessing the LLM generated solution for usability, d) determining whether a further attempt of querying the LLM should be made. This iterative process continues until either a

satisfactory solution is obtained or a threshold maximum number of allowed iterations is reached.

2. A set of eight quality attributes related to how easily a human user could use LLM generated code. These are accuracy, completeness, conciseness, clarity of logic, readability, well-structured-ness, parameter coverage and depth of explanation.
3. A set of three metrics that measure the user experience. The first is the average number of attempts, each of which is an iteration of the human-LLM interaction described above. The second is the average completion time for the task of coding using the LLM. The third is the success rate, where success means that useful code has been generated.

The authors illustrated the methodology using ChatGPT with 100 tasks in the programming language R. Usability was high: 3.8 out of 5, manually assessed on a Likert scale of 1 to 5 for each of the eight quality attributes. The average number of attempts was only 1.61 and the average completion time was 42 seconds.

Although subjective manual evaluation is valuable for the user-centred process proposed in the paper [17], it is labour-intensive and error-prone. Therefore, objective automated methods are preferable.

2.2. Automated Evaluation

As far as we know, the only way to use automation to evaluate code quality is via code smell detection. Siddiq et al. was perhaps the first to do this, in [18], where they used Pylint [19] to detect the code smells in three different training datasets: CodeXGlue [20], APPS [21], and Code Clippy [22]; Bandit [23] was also used in order to detect security code smells. To investigate the impact of code smell in training dataset, ten different code models, each based on the GPT-Neo 125M model, were trained on these three datasets, then tested on the HumanEval dataset [24] and compared with GitHub Copilot.

They found that the most frequent smells in the training datasets were also the most frequent in the generated code. They concluded that smells in the training datasets leaked into to the code models, although there was no statistical analysis of the correlations between the two, nor any causality analysis. However, their work has raised concerns about code smells in training datasets.

Moratis et al. applied code smell detection to the dataset DevGPT of reported iterative conversations in GitHub between developers and ChatGPT [25]. Two types of conversations were extracted:

1. *Write me this code*, with text instructions as input to produce a program code
2. *Improve this code*, with code snippets as input to improve the quality of the input program code

The conversations were then fed into the code smell detection tool PMD [26]. In the *Write me this code* category, there were 47 conversations and a total of 59 code smell violations in 144 code blocks. Half (50.8%) of the violations concerned the standard practices of code conventions, a third (37.3%) related

to styles of coding that have an impact on code readability and the remainder (11.9%) were violations of coding rules that were more likely to lead to errors.

In the *Improve this code* category, there were 334 conversations. In most cases, the output had fewer total violations and sometimes it was a lot fewer, suggesting that ChatGPT can be used for this purpose. Occasionally the output had more violations typically this was only one or two violations and not of the type that would introduce errors. Most conversations required fewer than 5 attempts; where more were needed it was usually because multiple code snippets were supplied as input.

Moratis et al. observed, however, that their findings were "*inherently optimistic, as it exclusively contains instances of successful interactions with ChatGPT*" [25]. Moreover, it is unclear whether the code quality is better or worse than that of human developers.

Another attempt to apply code smell detection to measure the ability of LLMs to improve the quality of existing code is due to DePalma et al. [27], who developed prompts to ask ChatGPT to refactor Java code to improve quality on 8 different quality attributes. Once again, PMD was applied both to the original code and the refactored code.

Liu et al.[28] took the idea of code smell measurement one step further to form a self-repairing mechanism. PMD was used once again for Java code but in conjunction with CheckStyle [29]. For Python code generation, the tools used were Pylint [19] and Flake8 [30].

They classified code quality issues into four categories: (a) Compilation and Runtime Errors, (b) Wrong Output (i.e. functional incorrectness of the generated code), (c) Code Style and Maintainability, and (d) Performance and Efficiency. For each of these categories, they identified the top 10 issues for Java and for Python. The dataset used was the LMDefect dataset [31] of 2033 coding tasks supplemented with coding tasks extracting from LeetCode. The experimental data shows great promise with a repair rate in the range 20% to 60%. However, fixes can often introduce new quality issues.

Table 1 summarises the related works mentioned in this subsection and contrasts them with the work reported in this paper. The column *Aims* gives the purpose of the research. The column *Usage* explains how code smell detection techniques achieve that purpose. The columns *Tools*, *Dataset*, *Language*, *LLM* and *Smell Types* give the code smell detection tool(s) used, the test dataset used to evaluate the LLM(s) with the size of the dataset in parentheses, the programming language in which the code is generated, the LLM(s) evaluated, and the types of code smell, respectively.

It is worth noting that the eight smells detected by DePalma et al. [27] are implementation smells. It is not explicitly stated what code smells were detected by Liu et al.'s work. However, we believe that architectural smells were not detected because there is no architectural level code generated by the test cases. For the same reason, architecture code smells were not detected in our work.

Table 1: Summary of Related Works

Work	Aims	Tools	Usage	Dataset (Size)	Language	LLM	Smell Types
Siddiq, et al. 2022	Investigating the impact of code smells in training datasets on the quality of generated code	Pylint	Detect code smells in training datasets and generated codes	HumanEval (164)	Python	GPT-Neo, GitHub Copilot	Implementation smells
		Bandit	Detect security smells in training datasets and generated codes				Security smells
Moratis, et al. 2024	Assessing the quality of code generated via iterative conversations in the <i>Write this code</i> scenario	PMD	Detect code smell and measure code quality	DevGPT (47)	JavaScript	ChatGPT	Best practice, code style, error prone
	Assessing the quality improvement of code generated via iterative conversations in the <i>Improve this code</i> scenario		Comparing code smells before and after refactoring	DevGPT (334)			
DePalma, et al. 2024	Evaluating LLM's capability of code refactoring	PMD	Assessing the quality of the code before and after refactoring	Ma et al.[32] (40)	Java	ChatGPT	Best practice, code style, design, documentation, error prone, multi-threading, performance, security
Liu, et al. 2024	Evaluating LLM's capability of fixing code quality issues	PMD, CheckStyle, Pylint, Flake8	Assessing the quality of generated code	LMDefect ⁺ (2033)	Java Python	ChatGPT	Implementation and design smells
This paper	Evaluating LLMs on various types of code smells for various types of code to generate and the complexities of coding task	PMD, CheckStyle, DesigniteJava	Assessing the quality of code generated in various scenarios	ScenEval (1000)	Java	Gemini Pro, ChatGPT, Codex, Falcon	Implementation and design smells

2.3. Research Questions

The existing works discussed above give some picture of the prevalence of code smells in LLM-generated code but their context and research questions are different from ours and there is no comparison with the human-written alternative. To bridge this research gap, we will ask the following open research questions:

- *RQ1*. Does LLM-generated code have a quality comparable to that of human-written code?

By human-written code, we mean code written by textbook authors or professional programmers, since we believe that can fairly represent the current best practice. Since our approach, outlined in Section 1.2, is to measure code smells, we will ask how does the incidence of smells in LLM-generated code compare with that of human-written code.

- *RQ2*. On which programming topics is LLM-generated code is weaker or stronger in quality compared to human-written code?

Once again, this can be rephrased in terms of code smells. How do code smells of LLM-generated vary with the question topic? More importantly, on which topics are the smells most worsened or most improved compared to human-written code?

- *RQ3*. How does the quality of LLM-generated code vary with the complexity of the coding task?

A closely related question would be is the difference in quality compared to human-written code greater for complex coding tasks? Both of these questions can be transformed into corresponding questions on code smells as above.

- *RQ4*. On which quality attributes is LLM-generated code worse compared to human-written code, since there is where research efforts could be directed?

We can rephrase this to ask which code smells are most prevalent in LLM-generated code and whether each smell is more or less common than in human-written code.

- *RQ5*. How is the correctness of LLM-generated code related to the usability of the code in terms of readability, modifiability, reusability and easiness to evolve?

To answer this question, we will separate the codes generated by LLMs according to their correctness, analyse their smells separately, and compare their code smells with the baseline.

3. Code Smell Detection

In this section, we will review the notion of code smells as background and explain how they can be detected. We will also explain how the code smell detection tools PMD, Checkstyle and DesigniteJava will be employed in our investigation.

3.1. The Notion of Code Smell

The concept of a *code smell* originated in Fowler's book on refactoring [12] having been coined by Beck [8]. It was defined as “*indications that there is trouble that can be solved by a refactoring*” and “*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*”. The authors described a list of 22 code smells, and how in each case, refactoring methods can help to improve the quality of the program. Since then, the notion of code smell has been intensively studied (see, for example, [13, 33, 11] for systematic literature reviews) and generalised to software smells [34].

Beck and Fowler noted two distinctive aspects of the notion of code smells. Firstly, they are *indicative* rather than “*precise criteria for flaws in program code*”. The code may not be flawed and may function correctly, but there may be future problems with maintenance, evolution, and reuse[11]. Secondly, they are *subjective*. It is better to judge smell based on “*informed human intuition*” and consequently, “*human agreement on smell detection is low*”, as has been proven by research [13].

The notion of code smell is linked to a number of other software engineering concepts and techniques, as follows:

- Smells are *indicators* or symptoms of a deeper design problem in the program code, as discussed above.
- Smells are suboptimal or poor solutions to a coding problem. Bad smells lead to a *technical debt* of needing to find better solutions later.
- Smells violate recommended *best practice* for the domain. These include coding conventions and/or software design principles. Therefore, smells can be detected by looking for the violations of best practices.
- Smells have a negative impact on the software *quality attributes* that are related to product revision and transition, such as modifiability, readability, testability, reusability, portability, etc. In this way, they make software difficult

to evolve, maintain, and reuse, and increase the likelihood of bugs, without themselves being bugs.

- As Fowler suggested, smells should and can be eliminated or reduced, for example, by *refactorings*, which are meaning-preserving transformations on the software.
- Smells are recurring problems in program code. The patterns of such recurring problems bear similarity to the notion of *anti-patterns*.

3.2. Types of Code Smells

Many types of code smells have been defined and investigated in the literature. They can be classified according to a number of different criteria, such as the effect caused, design principles violated, location of the smell, its granularity, etc [34]. In this paper, we adopt the classification proposed by Suryanarayana, Samarthayam and Sharma [9, 10] which distinguishes implementation smells from design smells (also known as micro-architectural smells) and architectural smells.

3.2.1. Implementation Smells

Implementation smells are concerned with suboptimal implementation choices that make the code unnecessarily complex, difficult to maintain, and harder to understand. We detect and analyse the following:

- *Inconsistent Naming Convention*. Deviations from the recommended naming conventions.
- *Excessive Complexity*. An expression, statement or a method is difficult to understand due to lack of clarity caused by excessive complexity. For example, a statement could be excessively long, an expression could be excessively nested and/or have too many operations, and a method could have too many lines of code, and/or has an excessive list of parameters.
- *Incompleteness*. A piece of code is unfinished with, for example, "TODO" or "FIXME" tags, or a statement is incomplete. For example, a catch block may be missing handling logic, a conditional construct may be missing a terminating else clause, a switch or selector statement may be missing a default case, or more generally, a block of code within curly braces { } contains no executable statements, etc.
- *Redundant Elements*. The presence of duplicate parameters, methods, or code blocks. A method or attribute may have an unnecessary modifier, such as `public` where that visibility is already implied, or `public static final` where `final` would have been enough.
- *Improper Alignment and Placement*. Code is not properly aligned according to coding standards, and/or an entity in the code is misplaced; for example, attributes may not be given in the recommended order.
- *Magic Number*. A numeric literal is used directly in code without being defined as a constant.

- *Dead Code*. Sections of code are no longer executed or provide no value for some other reason.
- *Resource Handling*. Inefficiencies in the use of resources.
- *Documentation*. Insufficient comments to explain the code properly.

3.2.2. Design Smells

Design smells are concerned with design choices, as presented in the program code, that violate fundamental design principles, such as poor use of object-orientation. They indicate the types of weaknesses that can lead to increased complexity, maintainability issues, and reduced code reusability. The following are the types of design smells defined by Suryanarayana et al. [9, 10]; these are all detected and analysed in this paper.

- *Abstraction Smell* – Issues related to improper, missing, or unnecessary abstractions, affecting code clarity and reusability.
- *Encapsulation Smell* – Violations of encapsulation principles, such as excessive exposure of internal details or inadequate access restrictions.
- *Modularisation Smell* – Poorly structured modules, including tightly coupled components, improper separation of concerns, and redundant dependencies.
- *Hierarchy Smell* – Problems in class hierarchies, such as deep inheritance trees, improper sub-classing, or lack of adherence to object-oriented principles.

3.2.3. Architectural Smells

Architectural smells are the weakness in the architectural design of the system, as presented in the code, that often lead to reduced system flexibility, modification difficulties and maintainability challenges. Typical examples include inappropriate layering and tight coupling between components and subsystems, etc. We will not consider these smells, however, because LLMs have limited capability for generating the entire system architecture and are not normally used for this purpose.

3.3. Detecting Code Smells

Code smell detection has been intensively studied in the software engineering literature; see, for example, [35, 36, 37] for systematic literature reviews. Fowler suggested that detection should be manual based on developer's experience and intuition. However, this is not scalable and repeatable. So automated tools should be used instead. Such tools can be classified into three types.

- *Static Code Analysis Approaches*.

Tools for static analysis are usually based on either metrics or pattern-matching. Metrics on program code include Lines of Code (LOC), Number of Attributes per Class (NOA), Number of Methods per Class (NOM), Number of Children Classes (NOC), Depth of Inheritance (DIT), etc. A *metrics-based* tool

detects code smells using a combination of these metrics. A draw-back of this approach is the arbitrary nature of the threshold values set for the metrics.

A *rule-based* tool, in contrast, defines a set of detection rules based on the syntactic structure of the code. Often these rules are linked to coding conventions and design principles. Sometimes the tool is configurable in that the smells can be specified with editable rules. Usually, they can be seamlessly integrated into existing development workflows. Typical examples of these tools include PMD, Stylecheck, Pylint, DesigniteJava, etc. Recently, such tools have been applied to LLM-generated code; see Section 2.

- *History-based Approaches.*

The evolution history of the system can be used to analyse the symptoms caused by smells and hence identify the smell. However, only a small number of smells can be detected this way.

- *ML-based Approaches.*

There are two approaches for using machine learning (ML) models [38, 39].

The first is to train a model with features based on metrics, like lines of code, cyclomatic complexity, coupling metrics, etc. This requires large, high-quality labelled datasets. However, these are rare for code smells. Consequently, such ML models have not achieved the performance suitable for practical use [38, 39]. For example, the Naive Bayes model reported in [40] has low F1-scores for most smells and low precision in particular, indicating a high number of false positives. Moreover, existing ML models for smell detection are binary classification models, i.e. each model only detects one type of smells.

The second approach is to use LLM models to detect code smells. However, a recent evaluation reported in [41] shows low F1 scores for both Llama variants and GPT-4, the latter below 0.04.

3.4. Use of Smell Detection Tools

In this paper, we will use static code analysis tools to detect code smells in both LLM-generated code and the reference solutions. As with Liu et al. [28], we use PMD [42, 26] and Checkstyle [43, 29]. They are based on widely recognised coding conventions: Google Java Style Guide¹ and the Sun Java Code Convention², respectively. Both of them are capable of detecting and reporting the violations of these rules. However, since both tools are relatively weak in detecting design level code smells, we also use DesigniteJava³, which detects smells according to the design principles violated.

Tables 2 and 3 show the smell detection rules provided by each tool used in our work. Columns *Tool Used* and *Detection*

Rules give the tool and the smell detection rule used. Readers are referred to the websites of the tools for the definitions of the rules.

Note that none of the tools cover all smells so we need to combine them for maximum coverage. One smell type may be detected by several different rules, even by different tools. The number of violations for such a smell type is calculated by summing up the numbers of violations of different rules.

Where a rule is implemented by more than one tool, however, the violation is counted only once. We have found in such cases that both tools give the same number of violations for the rule on the same code extract. In Table 2, such cases are indicated by a footnote reference⁴.

Since empirical studies have found it difficult to set a limit on the number of violations for the code still to be of good quality, we will count the number and compare it with a baseline; this reflects current practice.

4. Test System for Code Smell Analysis and Evaluation

Our experiments with LLMs need to be automated and we do this by applying the methodology of datamorphic testing proposed by Zhu et al. [14], [15]. This treats software testing as a systems engineering problem and it encourages both efficient management of test resources and the evolution of the test facilities alongside that of the software under test.

According to the methodology, a test system comprises two types of artefacts: *test entities* and *test morphisms*. The former are objects and documents involved in testing, such as test data, test datasets, test results, etc. while the latter are operations that manipulate and/or generate these entities to perform testing tasks. This methodology is supported by the test automation environment Morphy [15].

Morphy provides a Java framework in which a test system can be implemented as a Java class (more precisely, a hierarchy of Java classes) that consists of a set of attributes representing the test entities and a set of methods representing test morphisms. They are both annotated with metadata so that they can be recognised by Morphy, seamlessly integrated with Morphy's testing tools, and applied to achieve test automation. The following types of test morphisms are recognised by Morphy.

- *Seed Maker*: Generates initial test cases from other entities.
- *Datamorphism*: Transforms existing test cases into new ones.
- *Metamorphism*: Verifies the correctness of test cases and returns a Boolean result.
- *Test Set Filter*: Adds or removes test cases from a test set.

¹<https://google.github.io/styleguide/javaguide.html>

²<https://www.oracle.com/java/technologies/javase/codeconventions-introduction.html>

³<https://www.designite-tools.com/products-dj>

⁴The result from the tool by applying this smell detection rule is ignored because the same rule is already checked by another tool where the rule may have a different name. Only the result from one tool on the same rule is taken into account.

Table 2: Smell Detection Rules Used for Implementation Smells

Smell Name	Detection Rule(s)	Tool(s) Used
Inconsistent Naming Convention	Local Variable Naming Convention	PMD
	/Local Variable Name	/CheckStyle ⁽⁴⁾
	Formal Parameter Naming Convention	PMD
	Method Naming Convention	PMD
	/Method Name	/CheckStyle ⁽⁴⁾
	Class Naming Convention	PMD
	GenericsNaming	PMD
	AbbreviationAsWordInName	CheckStyle
	AbstractClassName	CheckStyle
	CatchParameterName	CheckStyle
	ConstantName	CheckStyle
	IllegalIdentifierName	CheckStyle
Excessive Complex	Simplify Boolean Expression	CheckStyle
	Simplify Conditional	PMD
	Simplify Boolean Return	PMD/CheckStyle
	Simplified Ternary	PMD
	Line Length	CheckStyle
	Method Length	CheckStyle
	/Long Method	/DesigniteJava ⁽⁴⁾
Redundancy	Excessive Parameter List	PMD/DesigniteJava ⁽⁴⁾
	Redundant Import	CheckStyle
	Redundant Modifier	CheckStyle
Incompleteness	Copy Paste Detector	PMD
	Missing Switch Default	CheckStyle
	Todo Comment	CheckStyle
Improper Alignment and placement	Empty Control Statement	PMD
	Empty Catch Block	PMD/CheckStyle ⁽⁴⁾
	EmptyBlock	CheckStyle
	Indentation	CheckStyle
	FileTabCharacter	CheckStyle
	NeedBraces	CheckStyle
	UselessParatheses	PMD
	LeftCurly	CheckStyle
	RightCurly	CheckStyle
	ParenPad	CheckStyle
Magic Number	MethodParamPad	CheckStyle
	Variable Declaration Usage Distance	CheckStyle
	Declaration Order	CheckStyle
Dead Code	Unused Formal Parameter	PMD
	Unused Local Variables	PMD/CheckStyle ⁽⁴⁾
	Unused Private Fields	PMD
	Unused Private Method	PMD
	Unused Imports	CheckStyle
Resource Handling	Close Resource	PMD
	Avoid Instantiating Objects In Loops	PMD
Documentation	Comment Required	PMD
	Comment Size	PMD
	Comment Content	PMD
	Javadoc Method	CheckStyle
	Javadoc Type	CheckStyle
	Missing Javadoc Package	CheckStyle
	Javadoc Variable	CheckStyle

- **Test Set Metric:** Maps a test set to a real value, such as test adequacy.
- **Test Case Filter:** Maps a test case to a Boolean value. It can be used to determine whether the test case should be retained in the test set.
- **Test Case Metric:** Assigns a real-valued metric to individual test cases (e.g., complexity).
- **Analyser:** Examines the test set and produces a test report.
- **Executor:** Runs the program under test using inputs from test cases and captures the outputs.

Given a test system implemented in Java, Morphy supports test automation at the following three levels.

- **Action:** Executes a single test activity using test morphisms, built-in functions or tools.

Table 3: Smell Detection Rules Used for Design Smells

Smell Name	Detection Rules	Tool Used
Modularity	God Class	PMD
	Data Class	PMD
	Too Many Methods	PMD
	Too Many Fields	PMD
	Use Utility Class	PMD
	Hide Utility Class Constructor	CheckStyle
	Broken Modularization	DesigniteJava
	Cyclically-dependent Modularization	DesigniteJava
	Hub-like Modularization	DesigniteJava
	Insufficient Modularization	DesigniteJava
	Law of Demeter	PMD
	Coupling Between Objects	PMD
	Class Fan Out Complexity	CheckStyle
Encapsulation	Visibility Modifier	CheckStyle
	Excessive Public Count	PMD
	Deficient Encapsulation	DesigniteJava
	Final Parameters	CheckStyle
	Final Class	CheckStyle
	Hidden Field	CheckStyle
Hierarchy	Unexploited Encapsulation	DesigniteJava
	Broken Hierarchy	DesigniteJava
	Cyclic Hierarchy	DesigniteJava
	Deep Hierarchy	DesigniteJava
	Missing Hierarchy	DesigniteJava
	Multipath Hierarchy	DesigniteJava
	Rebellious Hierarchy	DesigniteJava
	Wide Hierarchy	DesigniteJava
Abstraction	Dependency Cycles btw Packages	DesigniteJava
	Imperative Abstraction	DesigniteJava
	Multifaceted Abstraction	DesigniteJava
	Unnecessary Abstraction	DesigniteJava
	Untutilized Abstraction	DesigniteJava

- **Strategy:** Applies test strategies, which are algorithms with test morphisms and test entities as parameters.
- **Process:** Runs test scripts of a high-level of abstraction. Such scripts can be obtained by recording interactive operations of Morphy, which can be edited, or even manually written, and replayed.

Our test system extends that for a previous experiment, to analyse correctness and completeness of LLM-generated code [4] and uses the same benchmark, ScenEval. We define, however, a new test entity *code smell report* and the following new test morphisms.

- **Test Executors: LLM invokers.** Four test executors for each of the four LLM models *Gemini Pro*, *Codex*, *Falcon7B*, and *ChatGPT*. The last of these has been inherited from the previous work [4] but the first three are new. Each executor submits the query to its respective LLM via an API call, and then extracts the Java code from the response text and saves it to a file for further analysis.
- **Analysers: Code Smell Detector Invokers.** Three new test morphisms, *PMD-analyser*, *Checkstyle-analyser* and *DesigniteJava-Analyser*, which invoke the corresponding static code analysis tools PMD, Checkstyle and DesigniteJava, and save the code smell reports into files.
- **Test Set Metrics: Code Smell Statistical Analysers.** Three analysers, *Violations per Solution*, *Baseline Violations per Solution* and *Increase Rate to Baseline*, which perform statistical analysis on the code smell report files.

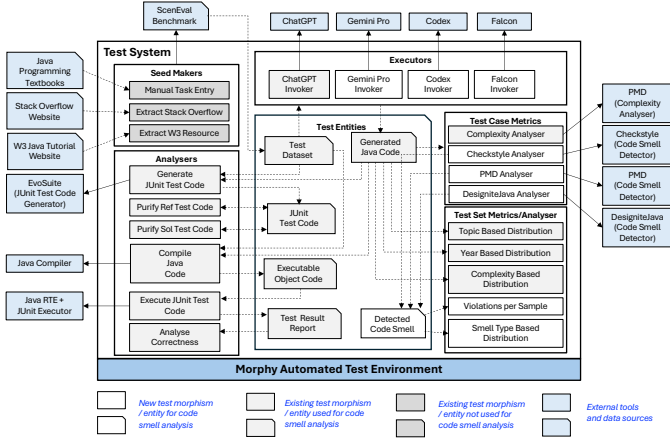


Figure 1: Structure of The Test System

Fig. 1 shows the structure of the test system. Test morphisms and entities inherited from, and explained in, our previous work [4] are shown in grey. New test morphisms and entities are shown in white. External tools invoked by them are shown in blue. Some of these are implemented in Python but invoked through a simple Python2Java interface.

5. Design of the Experiments

Before discussing the experiment process, we will first review the LLMs, the benchmark and the platform.

5.1. Subject LLMs

Table 4 presents basic information about the four state-of-the-art LLMs we have studied: *Gemini Pro*, *Falcon*, *ChatGPT*, and *Codex*. All are commonly used for software development.

Table 4: Large Language Models Evaluated

Name	Year	Version	Size
Gemini Pro	2023	Gemini Pro 1.0	Unknown
Falcon	2023	Falcon-7B	7B
ChatGPT	2023	GPT-3.5-turbo	Unknown
Codex	2021	GPT-3 (Codex)	12B

5.2. Benchmark

The benchmark ScenEval [4] contains more than 12,000 test cases of Java programming tasks. These test cases were curated from textbooks, online tutorial websites and the professional programming knowledge-sharing website Stack Overflow. In contrast to other benchmarks for code generation (see e.g. [16]), ScenEval has two distinctive features which make it ideal for our purpose.

1. Each test case is accompanied by the Java code for a reference solution, typically textbook answers and highly-rated Stack Overflow posts. As discussed in 1.2, their code quality represents the state of current practice so they enable us to establish a baseline of code smell for human-written Java code.

2. Each test case is also accompanied by metadata that specifies topic, complexity, source of the task, etc. This enables us to analyse the relationship between these concepts and code smell so that we can answer the research questions.

Our test dataset, sampled at random from the ScenEval benchmark, contains equal quantities (500 each) from textbooks and Stack Overflow. The other statistics are given in Table 5; Input Length and Complexity denote the number of words in the task description and the cyclomatic complexity of the reference solution.

Table 5: Statistical Characteristics of the Test Dataset

Feature	#Textbook Tasks	#Real Tasks
# Topics	25	18
# Tasks per Topic (average)	20.00	27.78
# Tasks per Topic (max)	79	61
# Tasks per Topic (min)	8	5
Input Length (average)	18.55	21.54
Input Length (max)	35	31
Input Length (min)	11	16
Complexity (average)	3.448	3.200
Complexity (max)	6	5
Complexity (min)	1	1
Number of Coding Tasks	500	500

5.3. Experiment Platform

The experiment was conducted with the automated test environment Morphy [14, 15] running on a desktop computer and is illustrated in Fig. 2. The LLM models under test were invoked through API calls as discussed in Section 4 and the smells of the LLM-generated code were analysed with PMD, Checkstyle and DesigniteJava as discussed in Section 3.

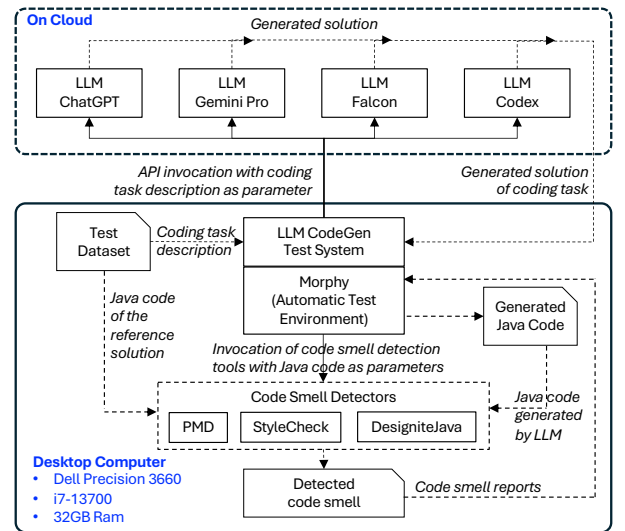


Figure 2: The Experiment Setup

5.4. Experiment Process

The process of the experiment consists of the following steps.

1. *Constructing Test Dataset*: A test dataset containing 1000 coding tasks was constructed by random sampling of the ScenEval benchmark.
2. *Analysing Smells of Reference Solutions*: For each coding task in the test dataset, the Java code for the reference solution is analysed by invoking the code smell detection tools PMD, Checkstyle and DesigniteJava. The reports on the violations of smell detection rules are saved into separate files for further statistically analysis to establish the baseline.
3. *Invoking LLMs*: For each coding task in the test dataset, the LLMs under test are queried through API invocations and the solutions returned were collected. The Java code in the returned text is separated from the explanation text and saved into a .java file.
4. *Analysing Smells of LLM Generated Code*: The Java codes generated by the LLMs are analysed via invocations of the code smell detection tools PMD, Checkstyle and DesigniteJava. The violations of the detection rules are saved into code smell report files.
5. *Analysing Correctness of Generated Code*: For each LLM generated Java code, the correctness of the code is determined by running test data on both the LLM-generated code and the reference solution; see [4] for details.
6. *Analysing Complexity of Generated Code*: The complexity of the LLM-generated code is measured the same way as in our previous work. [4].
7. *Statistical Analysis*: The code smell report files are parsed, and statistical analysis is performed on various subsets of the test dataset to answer the research questions. Each subset represents a different scenario in the use of LLMs. The smell detection rules are partitioned into subsets according to their type, where needed to answer a research question. Given a subset T of the coding tasks and a subset S of smell detection rules, the following statistical data are calculated.
 - (a) The number of violations of detection rules per solution, denoted by $VS_S^M(T)$, is calculated for each LLM M using Equ. (1).
 - (b) The baseline for the test subset T w.r.t. smell detection rules in S , i.e. the number of violations of smell detection rules per solution, denoted by $VS_S^B(T)$, is calculated from the code smell reports of reference solutions using Equ. (2).
 - (c) The increase rate of smells for LLM-generated code with respect to the baseline, denoted by $Inc_S^M(T)$, is calculated from $VS_S^M(T)$ and $VS_S^B(T)$ using Equ. (3).

These three equations are implemented as test set metrics and formally defined in the next subsection.

5.5. Metrics of Performance

Let t be a given coding task. We write $M(t)$ to denote the program code generated by a LLM model M on coding task t and $R(t)$ to denote the its reference solution in the benchmark.

Let s be any given smell detection rule and c be a given Java code sample. We write $V_s(c)$ to denote the set of violations of the rule s detected in the Java code c .

Let $T \neq \emptyset$ be a set of coding tasks, such as those for a specific topic or complexity in the test dataset.

Let $S \neq \emptyset$ be a set of smell detection rules, such as those for a particular type of code smell.

Definition 1. (LLM's Smell Violations Per Solution (VS))

We write $VS_S^M(T)$ to denote the smell violations of LLM M per solution w.r.t. a set S of smell detection rules and a set T of coding tasks, or simply violations per solution (VS). It is the average number of violations of the smell detection rules in S over the set of solutions generated by LLM M on coding tasks in T . Formally, we have that

$$VS_S^M(T) = \frac{\sum_{t \in T} \sum_{s \in S} \|V_s(M(t))\|}{\|T\|} \quad (1)$$

□

The corresponding calculation for the baseline is as follows.

Definition 2. (Baseline's Smell Violations per Solution)

We write $VS_S^B(T)$ to denote the baseline's smell violations per solution w.r.t. a set S of smell detection rules and a set T of coding tasks. This is the average number of violations of the smell detection rules in S over the reference solutions of the coding tasks in T . Formally, we have that

$$VS_S^B(T) = \frac{\sum_{t \in T} \sum_{s \in S} \|V_s(R(t))\|}{\|T\|} \quad (2)$$

□

Since the number of smell violations per solution is the only metric we are using to measure code smell, we will from now refer to it as the degree of code smell. Higher values mean poorer quality code. To compare the quality of LLM-generated code against a baseline, we will also measure the *increase rate* of code smells, as defined below.

Definition 3. (Increase Rate of Code Smells)

The increase rate of code smells for LLM model M with respect to the baseline on a set S of smell detection rules over a set T of coding tasks is denoted by $Inc_S^M(T)$, which is formally defined by the following equation.

$$Inc_S^M(T) = \frac{VS_S^M(T) - VS_S^B(T)}{VS_S^B(T)} \quad (3)$$

□

Positive values for this quantity mean that LLM-generated code is of lower quality than the reference solution.

6. The Results

In this section, we report the data collected from our experiments and answer each of the research questions with a statistical analysis of the data.

6.1. RQ1: Prevalence of Code Smells

Research question *RQ1* is concerned with the overall quality of the code generated by the LLMs. To answer this question, we calculated the VS on all smell detection rules over the whole test dataset for each LLM and compared it with the baseline. The results are shown in Fig. 3.

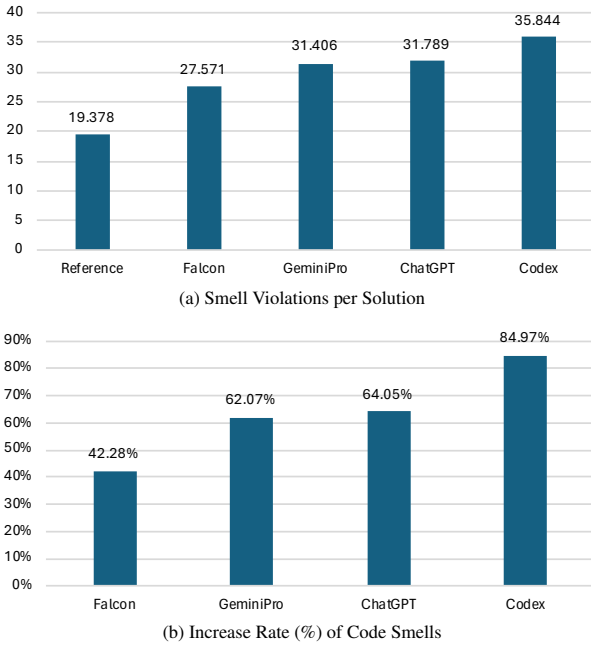


Figure 3: All Code Smells Detected on the Whole Test Dataset

As shown in Fig. 3(a), all four LLMs have stronger code smells than the baseline with Falcon performing the best (VS=27.571) and Codex the worst (VS=35.844). Moreover, the increase rate of code smells varies significantly in the range from 42.28% for Falcon to 84.97% for Codex. This is a strong evidence for the following observations.

Observation 1. *LLM-generated code is of poorer quality in terms of the smell violations per solution when compared to the human-written code.*

Observation 2. *The rate of increase in code smells of LLM generated code when compared to the human-written code varies significantly with the choice of LLM.*

6.2. RQ2: Variation of Code Smells by Topic

Research question *RQ2* aims to identify the types of tasks for which LLM-generated code is of poor quality so that improvement can be directed to these tasks. To answer this question, we divide up the test dataset according to the topic of the code to be generated and then calculate the VS on subsets of these coding tasks with all smell detection rules.

Table 6 presents the VS values for various programming topics and the increase rates for each LLM model.

From the experiment data, the following observations can be made.

First, the experiment data shows that LLMs performed differently on the program topics. Although the strength of code smells for the baseline also vary over code topics, the LLMs have a higher standard deviation on VS values when compared to the baseline (1.37). The standard deviations for LLMs are 1.90 for Falcon, 1.96 for ChatGPT, 2.08 for Codex and 2.14 for Gemini Pro, respectively. On average over all LLMs, the standard deviation is 1.85, which is an increase of 31.86% compared to the baseline. This indicates that LLMs have a robustness problem in maintaining consistency in code quality. In this respect, Falcon is the best (1.90) and Gemini Pro the worst (2.14).

Observation 3. *Over different coding topics, the degree of code smells in LLM generated code varies significantly more than human written code.*

Second, not only does the degree of code smells vary over different code topics, there is a pattern to the variation. We analysed the Pearson correlation coefficients between the baseline VS values and those of LLM generated code over various code topics. We found that there is a strong correlation between them. The Pearson correlation coefficients are 0.6652 for Falcon, 0.7249 for Gemini Pro, 0.7188 for ChatGPT and 0.7664 for Codex, respectively, and 0.7876 on average over all LLMs. Therefore, we have the following observation.

Observation 4. *The coding topics with strong code smells in human-written code are the same as the coding topics with strong code smells in LLM-written code.*

However, we found no strong correlation between the baseline VS values and the LLM's increase rates of code smells. In fact, the Pearson correlation coefficients are negative, between -0.1592 and -0.3808.

Finally, the data also shows that LLMs are more likely to worsen the code smells on more advanced coding topics. Table 7 shows the best and worst three topics as well as the most improved and worsened three topics by each LLM and on average over all studied LLMs. The best topics (i.e. the strength of code smells decreased) are *Basic Exercise*, *String*, *DateTime*; while the worst topics (i.e. the strength of code smells increased) are *Searching and Sorting*, *Encapsulation*, *Inheritance*, *Polymorphism*, *Interfaces*, and *Generics*.

There are a few topics on which LLMs improved the code quality in terms of code smells. These include *Regular Expression* by all LLMs and *Enum* improved by Gemini Pro, ChatGPT and Codex, *Collections* by Falcon and Gemini Pro, *Interfaces* and *Lambda* by Falcon and *Methods* by Codex, and *DataType* by ChatGPT. The highest improvement is 35.93% by Gemini Pro on the topic of *Regular Expressions*.

On average over all LLMs studied, the most worsened topics are *Encapsulation* by 138.53%, *Array* by 101.88%, and *OOP* by 101.88%. The largest increase rate of code smell is 165.38% by ChatGPT on the topic of *Encapsulation*. Thus, we have the following observations.

Table 6: Code Smell by Code Topics

Topic	Average Number of Voilations per Solution						Increase (%)				
	Baseline	Falcon	GeminiPro	ChatGPT	Codex	Average	Falcon	GeminiPro	ChatGPT	Codex	Average
Basic Exercise	1.96	1.99	2.91	2.77	2.35	2.51	1.53	48.47	41.33	19.90	27.81
DateTime	2.18	2.59	2.85	3.62	3.33	3.10	18.81	30.73	66.06	52.75	42.09
String	2.23	2.9	2.73	3.11	3.32	3.02	30.04	22.42	39.46	48.88	35.20
Array	2.39	3.78	3.99	5.88	5.65	4.83	58.16	66.95	146.03	136.40	101.88
Input Output	2.66	2.69	3.63	3.78	6.35	4.11	1.13	36.47	42.11	138.72	54.61
Lambda	2.71	2.41	3.95	4.16	4.33	3.71	-11.07	45.76	53.51	59.78	36.99
Collections	2.91	2.56	2.58	5.64	3.68	3.62	-12.03	-11.34	93.81	26.46	24.23
Recursive Methods	2.95	2.99	2.78	5.01	7.23	4.50	1.36	-5.76	69.83	145.08	52.63
Thread	2.96	3.57	4.59	5.74	6.22	5.03	20.61	55.07	93.92	110.14	69.93
Math	2.97	3.35	4.53	4.99	5.15	4.51	12.79	52.53	68.01	73.40	51.68
DataType	3.19	3.84	4.26	3.55	4.71	4.09	20.38	33.54	11.29	47.65	28.21
Methods	3.33	3.94	3.78	4.77	3.9	4.10	18.32	13.51	43.24	17.12	23.05
OOP	3.42	6.73	7.92	4.88	7.99	6.88	96.78	131.58	42.69	133.63	101.17
Exception Handling	3.45	3.22	5.44	5.39	5.31	4.84	-6.67	57.68	56.23	53.91	40.29
Encapsulation	3.64	9.91	7.95	9.66	7.21	8.68	172.25	118.41	165.38	98.08	138.53
Conditional	3.73	3.48	3.84	6.36	4.98	4.67	-6.70	2.95	70.51	33.51	25.07
Data Structure	4.34	4.26	4.76	5.23	8.12	5.59	-1.84	9.68	20.51	87.10	28.86
Enum	4.56	4.5	3.88	4.38	4.66	4.36	-1.32	-14.91	-3.95	2.19	-4.50
Generics	5.46	5.73	9.42	7.95	8.35	7.86	4.95	72.53	45.60	52.93	44.00
Interfaces	5.57	4.27	6.75	8.71	9.21	7.24	-23.34	21.18	56.37	65.35	29.89
Regular Expression	5.65	5.14	3.62	4.32	5.64	4.68	-9.03	-35.93	-23.54	-0.18	-17.17
Abstract Classes	5.66	6.31	7.62	7.69	7.28	7.23	11.48	34.63	35.87	28.62	27.65
Searching & Sorting	5.67	6.2	8.62	8.34	8.61	7.94	9.35	52.03	47.09	51.85	40.08
Polymorphism	5.72	6.91	6.74	8.44	8.48	7.64	20.80	17.83	47.55	48.25	33.61
Inheritance	6.91	7.59	8.96	8.98	10.48	9.00	9.84	29.67	29.96	51.66	30.28
Average	3.85	4.43	4.93	5.73	6.10	5.30	15.22	28.01	48.98	58.53	37.69
Std dev	1.37	1.90	2.14	1.96	2.08	1.85	39.58	37.24	39.53	41.59	31.86

Table 7: The Best/Worst Topics And The Most Improved/Worsened Topics

Model	Best Topics (VS)	Worst Topics (VS)	Most Improved Topics (Inc %)	Most Worsened Topics (Inc %)
Baseline	Basic Exercise (1.96) DateTime (2.18) String (2.23)	Searching & Sorting (5.67) Polymorphism (5.72) Inheritance (6.91)	N/A N/A N/A	N/A N/A N/A
Falcon	Basic Exercise (1.99) Lambda (2.41) Collections (2.56)	Polymorphism (6.91) Inheritance (7.59) Encapsulation (9.91)	Interfaces (-23.34) Collections (-12.03) Lambda (-11.07)	Array (58.16) OOP (96.78) Encapsulation (172.25)
Gemini Pro	Collections (2.58) String (2.73) Recursive Methods (2.78)	Searching & Sorting (8.62) Inheritance (8.96) Generics (9.42)	Regular Expression (-35.93) Enum (-14.91) Collections (-11.34)	Generics (72.53) Encapsulation (118.41) OOP (131.58)
ChatGPT	Basic Exercise (2.77) String (3.11) DataType (3.55)	Interfaces (8.71) Inheritance (8.98) Encapsulation (9.66)	Regular Expression (-23.54) Enum (-3.95) DataType (11.29)	Thread (93.92) Array (146.03) Encapsulation (165.38)
Codex	Basic Exercise (2.35) String (3.32) DateTime (3.33)	Searching & Sorting (8.61) Interfaces (9.21) Inheritance (10.48)	Regular Expression (-0.18) Enum (2.19) Methods (17.12)	Array (136.40) Input Output (138.72) Recursive Methods (145.08)
Average	Basic Exercise (2.51) String (3.02) DateTime (3.10)	Searching & Sorting (7.94) Encapsulation (8.68) Inheritance (9.00)	Regular Expression (-17.17) Enum (-4.50) Methods (23.05)	OOP (101.17) Array (101.88) Encapsulation (138.53)

Observation 5. *The LLM-generated code has the best quality on basic coding topics, and the worst on advanced coping topics.*

Observation 6. *The LLM-generated code can have better quality in comparison with human written code on certain coding topics, while it can also have significantly worse code quality, especially on advanced coding topics.*

6.3. RQ3: Variation of Code Smells by Complexity

Research question RQ3 is concerned with how code quality varies with the complexity of the coding tasks. To answer this question, we calculated the VS on subsets of test cases that were formed according to the complexity of the coding tasks. Here, the complexity of a coding task is measured on the complexity of the reference solution provided by the benchmark ScenEval.

Three different complexity metrics were used: cyclomatic complexity, cognitive complexity and lines of code. The results are presented in Fig. 4 graphically.

From Fig. 4, it can be seen clearly that the VS tends to increase with each of three different metrics of complexity. This is confirmed by the Pearson Correlation coefficients between VS and complexity; see Table 8. For cyclomatic complexity, the Pearson correlation coefficients are all strongly positive (above 0.9) for each LLM as well as the baseline. For lines of code, the coefficients are a bit lower but still very high (in the range between 0.8685 and 0.9952 except ChatGPT (0.6347). For cognitive complexity, the results are mixed: very strong for Falcon (0.9754), but much lower for ChatGPT (0.3025), and around 0.6 for Gemini Pro and Codex, and 0.6894 on average over all LLMs.

Therefore, we have the following observations.

Table 8: Correlations Bwt VS and Coding Task Complexities

Complexity	Baseline	Falcon	GeminiPro	ChatGPT	Codex	Average
Cyclomatic	0.9344	0.9555	0.9916	0.9485	0.9962	0.9653
Cognitive	0.8800	0.9754	0.6312	0.3025	0.6578	0.6894
Lines of Code	0.9952	0.8694	0.9532	0.6347	0.8900	0.8685

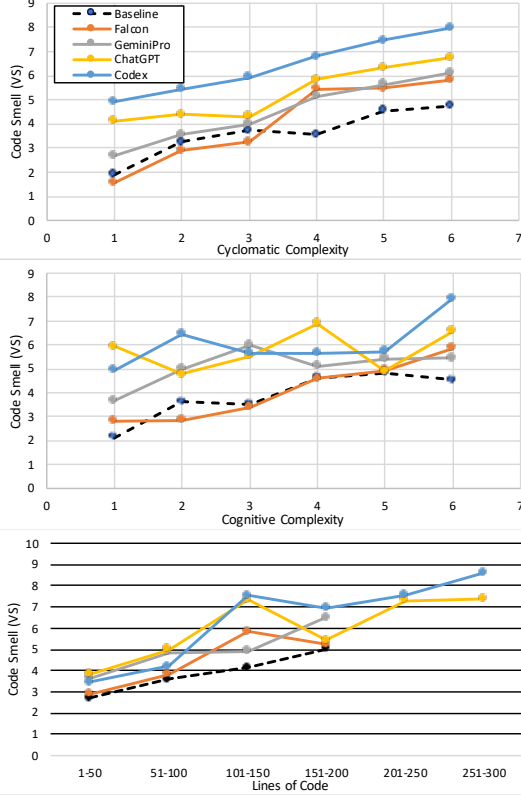


Figure 4: Variation of Code Smells by Complexity

Observation 7. For both human written code and LLM generated code, the strength of code smells increases with the complexity of coding tasks.

It is worth noting that the baseline VS value also increases with cyclomatic complexity but it does so at a slower rate than any of the four LLMs. So, we can see that LLMs tend to struggle to produce good quality code when they are given highly complex tasks.

However, by analysing the increase rate of code smells in the code generated by LLMs with respect to human written code, we found no obvious link between the complexity of coding tasks to the increase rate of code smells as shown in Fig. 5.

Therefore, we have the following observation.

Observation 8. There is no clear evidence that the increase rate of code smells in LLM generated code is correlated to the complexity of coding task.

6.4. RQ4: Variation of Code Smells by Smell Types

Research question RQ4 aims to identify the specific quality issues in LLM generated codes. We calculated the VS for each

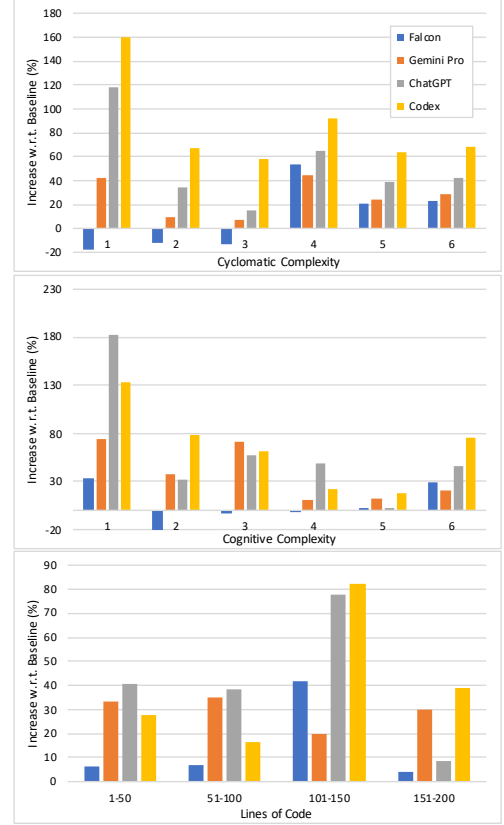


Figure 5: Increase Rates of Code Smells by Complexity

specific type of code smell on the whole test dataset and compared with the baseline.

Table 9 presents, in the form of a heat map, the VS for each specific type of smells as well as the increase rates for each LLM model in comparison with the baseline. The highest VS scores and the largest (i.e. worst) increase rate are highlighted in red, while the lowest VS scores and lowest increase rates are coloured in blue. Implementation smells are listed in the top half and design smells in the bottom half.

From the experiment data, we observed the following phenomena.

Observation 9. The least prevalent types of implementation smells for all LLMs as well as the reference solutions are *Incompleteness*, *Inconsistent Naming Convention* and *Redundancy*.

Observation 10. The worst types of implementation smells for all LLMs and human written code are *Magic Number*, *Documentation* and *Improper Alignment and Placement*.

In general, there is a very strong correlation between LLM generated code and human written code on the VS scores on the types of code smells. As shown in Table 10, for implementation types of code smells, the Pearson correlation coefficients between the baseline and the code generated by each LLM are all very close to 1. For design code smells, the Pearson correlation coefficients are in the range between 0.7263 for Falcon and

Table 9: Code Smells by Smell Type

Smell Type	Average Numbers of Violations per Solution (VS)						Increase in VS (%)				
	Reference	ChatGPT	GeminiPro	Falcon	Codex	Average	ChatGPT	GeminiPro	Falcon	Codex	Average
Incompleteness	0.0030	0.0140	0.0160	0.0080	0.0150	0.0133	366.67	433.33	166.67	400.00	341.67
Inconsistent Naming Convention	0.0040	0.0500	0.0510	0.0260	0.0500	0.0443	1150.00	1175.00	550.00	1150.00	1006.25
Redundancy	0.0180	0.0710	0.0780	0.0410	0.0710	0.0653	294.44	333.33	127.78	294.44	262.50
Dead Code	0.0860	0.0580	0.0640	0.0860	0.0560	0.0660	-32.56	-25.58	0.00	-34.88	-23.26
Resource Handling	0.5650	0.5840	0.6420	0.5650	0.5950	0.5965	3.36	13.63	0.00	5.31	5.58
Excessive Complex	0.6530	1.1510	1.3530	0.6610	1.2740	1.1098	76.26	107.20	1.23	95.10	69.95
Magic Number	0.8510	1.3740	1.5230	0.8570	1.5460	1.3250	61.46	78.97	0.71	81.67	55.70
Documentation	2.3140	3.2500	3.3170	2.3140	3.2810	3.0405	40.45	43.34	0.00	41.79	31.40
Improper Alignment / Placement	11.1500	20.4990	19.5600	19.1890	24.1850	20.8583	83.85	75.43	72.10	116.91	87.07
Average of Impl Smells	15.6440	27.0510	26.6040	23.7470	31.0730	27.1188	72.92	70.06	51.80	98.63	73.35
StDev of Impl Smells	3.6049	6.6429	6.3199	6.2491	7.8484	6.7634	370.61	379.28	179.61	368.75	324.12
Hierarchy	0.0000	0.0020	0.0020	0.0020	0.0020	0.0020	N/A	N/A	N/A	N/A	N/A
Abstraction	0.0000	1.0270	1.0380	1.0630	1.0350	1.0408	N/A	N/A	N/A	N/A	N/A
Modularity	1.8240	1.9180	1.9460	1.5440	1.9380	1.8365	5.15	6.69	-15.35	6.25	0.69
Encapsulation	1.9100	1.7910	1.8160	1.2150	1.7960	1.6545	-6.23	-4.92	-36.39	-5.97	-13.38
Average of Design Smells	3.7340	4.7380	4.8020	3.8240	4.7710	4.5338	26.89	28.60	2.41	27.77	21.42
StDev of Design Smells	1.0785	0.8811	0.8939	0.6669	0.8873	0.8276	8.05	8.21	14.88	8.64	9.94
Average of All Smells	19.3780	31.7890	31.4060	27.5710	35.8440	31.6525	64.05	62.07	42.28	84.97	63.34
StDev of All Smells	3.0172	5.5117	5.2477	5.1768	6.5146	5.6109	82.67	73.92	71.57	115.91	85.96

Table 10: Pearson Correlation Coefficients btw Baseline and LLMs' VS Scores on Various Smells Types

	ChatGPT	GeminiPro	Falcon	Codex	Average
Impl Smells	0.9987	0.9990	0.9961	0.9973	0.9980
Design Smells	0.8757	0.8766	0.7263	0.8748	0.8506

0.8766 for Gemini Pro. Therefore, we can confidently conclude that:

Observation 11. *The prevalence of code smells in LLM-generated code on various smell types is strongly correlated with that of human-written code.*

The experiment data also demonstrated that the prevalence of a type of code smell in human written code does not imply that the smell increases in LLM generated code. As shown in Table 11, for implementation smells, the Pearson correlation coefficients between LLMs' smell increase rates and the VS scores of the baseline are all negative in the range between -0.2549 for Gemini Pro and -0.1518 for Falcon, where the average over all LLMs is -0.2188.

Table 11: Pearson Correlation Coefficients btw Baseline VS Scores and LLMs' Increase Rates on Various Smells Types

	ChatGPT	GeminiPro	Falcon	Codex	Average
Impl Smells	-0.2257	-0.2549	-0.1518	-0.2064	-0.2188
Design Smells	-1	-1	-1	-1	-1

However, it is observable that the largest increase rates of code smells are on the types that are the least prevalent smell types of the baseline.

Observation 12. *The largest increases of smells in LLM generated code happened at the smell types of Incompleteness, Inconsistent Naming Convention and Redundancy.*

Finally, the Pearson correlation coefficients between LLMs' increase rates over various types of implementation smells are

all very close to 1; See Table 12. This implies the following observation.

Observation 13. *LLMs consistently increase the strength of code smells over various types of code smells.*

Table 12: Pearson Correlation Coefficients btw LLMs' Increase Rates of Implementation Smells

	ChatGPT	GeminiPro	Falcon	Codex
ChatGPT				
Gemini Pro	0.9985			
Falcon	0.9931	0.9884		
Codex	0.9992	0.9984	0.9926	

From the experiment data, we can also have the following observations.

Observation 14. *All LLMs performed well on the encapsulation type of design smells in comparison with the baseline.*

Observation 15. *LLMs' performance on design smells vary significantly with increase rates ranging from 2.41% for Falcon to 28.60% for Gemini Pro, and the average increase rate over all LLMs is 21.42%. Falcon performed better on design smells than the other LLMs.*

Finally, by analysing the distributions of VS scores for each types of smells, we have the following observation.

Observation 16. *For each type of smells, the violations of smell detection rules are concentrated in a small number of specific smells.*

Table 13 lists the most prevalent smells in each smell type, where column *Top Smell(s)* lists the most prevalent smell(s) of the smell type given in the column *Smell Type*. Column *#Violations* gives the average numbers of violations of the specific smell rule over all LLMs. Column *Weight* gives the ratio of the

Table 13: The Most Prevalent Smells in Each Type

Smell Type	Top Smell(s)	#Violations	Weight (%)	Baseline
Inconsistent Naming Convention	Abbreviation As Word In Name	44.25	100.00	4
Excessive Complex	Line Length	1105.25	99.59	653
	Simplify Boolean Expression	3.25	0.29	0
Redundancy	Redundant Modifier	59.5	91.19	18
	Redundant Import	5.75	8.81	0
Improper Alignment / Placement	Indentation	20732	99.39	11119
	FileTabCharacter	46	0.22	0
	Variable Declaration Usage Distance	39.75	0.19	6
Incompleteness	Missing Switch Default	10.25	77.36	2
	Empty Catch Block	1.5	11.32	1
Magic Number	Magic Number	1325	100.00	851
Dead Code	Unused Imports	66	100.00	86
Resource Handling	Close Resource	572.5	95.98	539
	Avoid Instantiating Objects In Loops	24	4.02	26
Documentation	Comment Required	2705	88.97	2103
	Javadoc Variable	224.25	7.38	166
	Comment Size	98.5	3.24	39
Modularity	Use Utility Class	917.75	49.97	912.00
	Hide Utility Class Constructor	918.75	50.03	912.00
Encapsulation	Final Parameters	1490.25	90.07	1730.00
	Hidden Field	118.50	7.16	114.00
Hierarchy	Broken Hierarchy	2.00	100.00	0.00
Abstraction	Unutilized Abstraction	1036.50	99.59	0.00
	Imperative Abstraction	3.75	0.36	0.00

violations over all smells of the type. Column *Baseline* gives the number of violations in the reference solutions.

Note that there are fewer violations of design smells than implementation smells. We believe that there are two reasons for this. First, the test dataset contains very few coding tasks where the solutions require a large number of classes. In fact, only 68 (6.8% of the dataset) require more than one class. Design smells like Hierarchy smells do not present in code that has only one class. Second, there are less detection rules for design smells than those for implementation smells. However, fewer violations of design smells do not imply the better design quality because design smells are at a higher level of abstraction and of greater granularity. Each violation of design smell could have a more serious impact than one violation of an implementation smell. It is not meaningful to compare the number of violations of design smells to that of implementation smells.

6.5. RQ5: Variation of Code Smells by Correctness

Research question RQ5 aims to understand how the correctness of LLM generated code relate to code smells.

To determine the correctness of a LLM generated solution, test cases were generated from both the reference solution and the LLM-generated code by employing the EvoSuite tool. These two sets of test cases are merged into one test suite. Both the reference solution and the generated code are tested on the test suite. If the reference solution fails on a test case that is generated from the LLM generated code, a commission error is detected. If the generated code fails on a test case that is generated from the reference solution, an omission error is detected. If neither a commission error nor an omission error are detected, i.e. it passes all of the tests, we regard the LLM generated code is correct. Readers are referred to [4] for details about how this is conducted.

Fig. 6 shows the number of LLM generated programs that pass all tests. From the data shown in the figure, Gemini Pro performed the best with a success rate of 74.3% passing all tests, while Falcon is the worst with a success rate only 47.0%. Codex and ChatGPT performed very similar, both have a success rate around 68.0%.

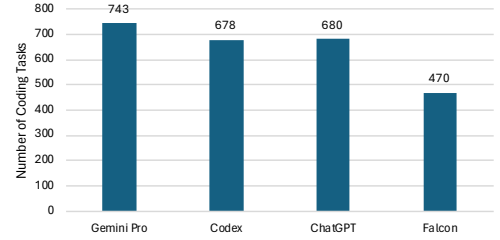


Figure 6: Numbers of LLM Generated Solutions Passed Test

To analyse how correctness is related to code smell, we split the test dataset into two subsets: one contains coding tasks that the LLM generated a correct code; the other contains tasks that LLM failed to generate a correct model. The smell violations per solution are calculated for each subset on all code smell detection rules. The results are shown in Table 14.

Table 14: Smells of Correct and Incorrect Codes

LLM	All Smells		Impl Smells		Design Smells	
	Correct	Incorrect	Correct	Incorrect	Correct	Incorrect
ChatGPT	31.33	32.93	26.63	28.11	4.70	4.82
Gemini Pro	30.63	33.88	25.86	29.00	4.77	4.88
Falcon	32.05	23.68	27.87	20.17	4.18	3.51
Codex	35.64	36.44	30.87	31.66	4.77	4.78
Average	32.41	31.73	27.81	27.24	4.60	4.50

From Table 14, we can observe that correct code has less smells than incorrect code on average for three out of four LLMs: Gemini Pro, Codex and ChatGPT. However, Falcon is an exception. Its incorrect code has less smells than its correct code. Table 15 shows the rates of increase (%) in code smells from correct code to the incorrect code for different LLMs.

Table 15: Increases (%) in Smells from Correct to Incorrect Codes

LLM	All Smells	Impl Smells	Design Smells
ChatGPT	5.13	5.57	2.63
Gemini Pro	10.62	12.16	2.29
Falcon	-26.11	-27.63	-15.93
Codex	2.23	2.55	0.17

From Table 15, we have the following observation.

Observation 17. *The degree of differences between the correct and incorrect codes in terms of the strength of code smells varies significantly with the LLM models. For some LLMs, incorrect codes are more smelly; while for the others, the opposite is observable.*

7. Discussion on Threats to Validity

In this section, we discuss the potential threats to the validity of the experiment reported in the previous section and how these threats were addressed in the design and conduct of the experiment. We will also discuss how to further reduce the threats in future work. We will apply Wohlin et al.’s the framework [44] of classifying the threats to validity in software engineering experiments, as it is among the frameworks most used by the researchers in software engineering.

7.1. Construct Validity

Construct validity is concerned with whether the data obtained by measurement and observation correctly and adequately represent the abstract concepts under study. In our context, it means whether the code smell detection rules correctly, adequately and fairly represent the quality aspects related to the readability, maintainability, ease of evolution, etc.

One primary threat to this validity lies in the reliance on the accuracy and coverage of the static analysis tools used in our study. Any limitations or inaccuracies in these tools could affect the precision of our smell detection. To mitigate this threat, we selected widely used and validated tools (i.e., PMD, Checkstyle, and DesigniteJava) that have demonstrated reliability in prior research and practice.

Also, we focused solely on code smells detectable by PMD, Checkstyle, and DesigniteJava. While this may exclude certain types of smells, the selected set of detection rules represents a well-established and widely adopted list of code smells. These have been well documented and widely used both in academic research and industry practice, lending credibility to their relevance and maturity. Moreover, we have combined the smell detection rules provided by these tools to maximised the coverage of the smells.

7.2. External Validity

The external validity is concerned with to what extent the results of an experiment can be generalised. A potential threat to external validity involves the generalisability of our findings to other LLMs not studied in our experiments, coding in programming languages other than Java, and those types of coding tasks not covered by the test dataset.

Our analysis is based exclusively on Java programs from the ScenEval dataset, which may limit the applicability of the results to code generation tasks in other programming languages.

Additionally, we evaluated outputs from some of the most widely used generative models GeminiPro, ChatGPT, Codex, and Falcon in Table 4. Other versions of these ML models and other ML models, such as CodeBERT [45] and CodeT5 [46], were not studied. Therefore, the results may not generalise across all generative coding models. However, there are observations that are consistent on all LLMs that we studied. These observations should be able to generalise to other ML models.

Our dataset covers a wide range of topics. However, the majority of coding tasks are of small scale in terms of complexity. Moreover, very few of the coding tasks require multiple classes.

The conclusions drawn from our experiment should be limited to the coding tasks well represented by our dataset. Any generalisation of our conclusions to other kinds of coding tasks should be used with great care.

7.3. Internal Validity

Internal validity is concerned with the appropriateness of the design and conduct of the experiments. A typical example of the threats to internal validity is the existence of factors that influence the causal relationships under study but are not measured and are not under our control.

A potential threat to the internal validity of our experiment is that LLMs are inherently nondeterministic. For this reason, we have selected a large number of test cases (1000) at random to minimise the impact of LLM’s randomness. The scale of our experiment is much larger than most of the studies of LLMs’ capability in code generation. For future research, this threat to internal validity can be further reduced by using even more test cases and repeating the invocations of LLMs on each coding task.

Another potential threat to internal validity is that the quality of program code in general and code smells in particular are very subjective as we discussed in Section 1 and 3. We have addressed this threat at the methodology level by excluding human factors from the experiment by using a baseline formed by professionally written code and at the technology level by employing the quantitative analysis of the experiment data using objective metrics.

Finally, a potential threat to internal validity is that the implementation of the test system may contain bugs, thus the data collected may have errors. We have addressed this threat by careful testing and debugging of the test system. Moreover, to ensure the experimental reproducibility, the source code of the test system as well as the data are available to the public in the GitHub repository ⁵.

7.4. Conclusion Validity

Conclusion validity is concerned with whether the conclusion drawn from the experiment is logically valid, such as whether correct statistical inference methods are used properly and whether the statistical inference power is strong enough.

Due to the fact that the experiments with LLMs are time consuming and resource demanding, the statistical inference power in this work is not ideal because the scale of our experiment is still limited. However, it is already much larger than other existing similar works. We believe it is not a serious threat to conclusion validity. For future work, repeating the experiments with a larger test dataset will further reduce the threat.

8. Conclusion and Future Work

In this paper, we proposed a scenario-based methodology to evaluate the usability of LLM-generated code on readability,

⁵URL: <https://github.com/hongzhu6129/EvaluateLLMCodeSmell>

modifiability, reusability, ease of maintenance, ease of evolution, etc., through assessing the code smells and comparing with a baseline obtained from code written by professional programmers. An automated test system is designed and implemented in the datamorphic testing method. An intensive experiment with four prominent LLMs is conducted using the ScenEval benchmark for generating Java code.

We have found that the code smells, measured by the average number of violations of code smell detection rules per solution, is significantly greater in LLM-generated code compared to human-written code. Across all LLMs, the average increase rates of implementation and design smells are 73.35% and 21.42%, respectively, while the average increase rate over all smells is 63.34%.

The performances of LLMs vary significantly over different topics of coding tasks and smell types. In general, the more complicated a coding task is, the stronger is the smell in LLM-generated code. The types of code smells that are the strongest in human written code are also the most prevalent in LLM generated code. However, the increase rates of code smell types in LLM generated code show no correlation to the prevalence of the type of code smell in human written code. In general, the quality of generated code decreases with the complexity of the code task. This correlation is very clear when coding task complexity is measured by cyclomatic complexity and the lines of code, but is slightly less clear with cognitive complexity.

For future work, it is worth further expanding and repeating the experiments with more LLM models and using larger test dataset to reduce the risks of the potential threats to validity as discussed in Section 7.

As discussed in Section 3, it is difficult to set a threshold on the number of violations for the code to be of acceptable quality. We could only compare with the baseline number, which reflects current practice. Hence, from our experiment data, we have difficulty to answer the question: is the quality of LLM generated code acceptable for use? We encourage further research in this area.

Moreover, LLM-generated code has broader quality aspects of usability that are worth considering, for example, security and runtime efficiency of the generated code, which is addressed by a recent work by Jonnala et al. [47]. This way we hope to provide a more holistic evaluation of LLM performance for code generation.

Finally, it is worth investigating how code smell detection can be used to improve the quality of LLM generated code in a multi-attempt process proposed by Miah and Zhu [17].

Acknowledgement

The research work reported in this paper is partly funded by Google's PaliGemma Academic Program, which provided credits for using Google Cloud Platform resources.

References

- [1] I. Shani, Survey reveals AI's impact on the developer experience, GitHub Blog, <https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/> (2024).

- [2] D. DeBellis, K. M. Storer, A. Lewis, B. Good, D. Villalba, E. Maxwell, K. Castillo, M. Irvine, N. Harvey, Accelerate state of DevOps, DORA 2024 Report, Google Cloud. <https://cloud.google.com/blog/products/devops-sre/announcing-the-2024-dora-report> (2024).
- [3] W. Harding, AI copilot code quality: Evaluating 2024's increased defect rate via code quality metrics, Tech. Rep., Alloy.dev Research, 211m Lines of Analyzed Code + Projections for 2025, (Feb. 2025).
- [4] D. G. Paul, H. Zhu, I. Bayley, ScenEval: A benchmark for scenario-based evaluation of code generation, in: Proc. of 2024 IEEE Int'l Conf. on Artificial Intelligence Testing (AITest 2024), July 2024, pp. 55–63.
- [5] A. Zdravkovic, AMD takes holistic approach to AI coding copilots: The chipmaker is using ai throughout the software-development life cycle, IEEE Spectrum 62 (6), pp26–31, (2025)
- [6] B. K. Deniz, C. Gnanasambandam, M. Harrysson, A. Hussin, S. Srivastava, Unleashing developer productivity with generative AI, McKinsey Digital, 7 (2023).
- [7] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, E. Aftandilian, Measuring github copilot's impact on productivity, Communications of the ACM, 67 (3) pp.54–63, (2024).
- [8] K. Beck, M. Fowler, Bad smells in code, Ch. 3, in: [12], pp. 75–88.
- [9] G. Suryanarayana, G. Samarthiyam, T. Sharma, Refactoring for Software Design Smells: Managing Technical Debt, Elsevier Science & Technology, 2014.
- [10] T. Sharma, M. Fragkoulis, D. Spinellis, Does your configuration code smell?, in: Proc. of 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR 2016), IEEE, Athens, Greece, 2016, pp. 189–200.
- [11] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, Code smells and refactoring: A tertiary systematic review of challenges and observations, Journal of Systems and Software, 167, 110610, (2020)
- [12] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.
- [13] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. Do Nascimento, M. F. Freitas, M. G. De Mendonça, A systematic review on the code smell effect, Journal of Systems and Software, 144 pp450–477, (2018)
- [14] H. Zhu, D. Liu, I. Bayley, R. Harrison, F. Cuzzolin, Datamorphic testing: A method for testing intelligent applications, in: 2019 IEEE International Conference On Artificial Intelligence Testing (AITest 2019), IEEE, 2019, pp. 149–156.
- [15] H. Zhu, I. Bayley, D. Liu, X. Zheng, Automation of datamorphic testing, in: Proc. of 2020 IEEE Int'l Conf. On Artificial Intelligence Testing (AITest 2020), pp. 64–72, (2020)
- [16] D. G. Paul, H. Zhu, I. Bayley, Benchmarks and metrics for evaluations of code generation: A critical review, in: Proc. of 2024 IEEE International Conference on Artificial Intelligence Testing (AITest 2024), pp. 87–94 (2024).
- [17] T. Miah, H. Zhu, User centric evaluation of code generation tools, in: 2024 IEEE International Conference on Artificial Intelligence Testing (AITest 2024), pp. 109–119, IEEE, (2024)
- [18] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, J. C. S. Santos, An empirical study of code smells in transformer-based code generation techniques, in: Proc. of 2022 IEEE 22nd Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM 2022), IEEE, USA, 2022.

- [19] S. Thénault, et al., Pylint: Code analysis for Python. Accessed: 2025-04-11 (2001).
- [20] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al., Codexglue: A machine learning benchmark dataset for code understanding and generation, arXiv preprint arXiv:2102.04664 (2021).
- [21] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, et al., Measuring coding challenge competence with apps, arXiv preprint arXiv:2105.09938 (2021).
- [22] N. Cooper, A. Arutiunian, S. Hincapié-Potes, B. Trevett, A. Raja, E. Hossami, M. Mathur, et al., Code clippy data: A large dataset of code data from GitHub for research into code language models (Oct. 2021). URL :<https://github.com/CodedotAI/gpt-code-clippy/wiki/Dataset>
- [23] Bandit, Bandit: Security linter for Python code, <https://bandit.readthedocs.io/>, accessed: 2025-04-11.
- [24] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).
- [25] K. Moratis, T. Diamantopoulos, D.-N. Nastos, A. Symeonidis, Write me this code: An analysis of chatgpt quality for producing source code, in: Proc. of the 21st IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR 2024), Thessaloniki, Greece, IEEE/ACM, 2024.
- [26] T. Copeland, PMD Applied, Vol. 10, Centennial Books, San Francisco, 2005.
- [27] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, E. A. Al Omar, Exploring ChatGPT's code refactoring capabilities: An empirical study, Expert Systems with Applications, 249 (Part B) 123602, (2024)
- [28] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X. D. Le, D. Lo, Refining ChatGPT-generated code: Characterizing and mitigating code quality issues, ACM Transactions on Software Engineering and Methodology, 33 (5) pp.1–26, (2024)
- [29] O. Burn, Checkstyle, <http://checkstyle.sourceforge.net/>, accessed: 2025-04-11 (2003).
- [30] I. Cordasco, T. Ziade, Flake8: Your tool for style guide enforcement, Programa de computador, accessed: 2025-04-11 (2010).
- [31] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, S. H. Tan, Automated repair of programs from large language models, in: Proc. of 2023 IEEE/ACM 45th Int'l Conf. on Software Engineering (ICSE 2023), pp. 1469–1481, IEEE, 2023.
- [32] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, Y. Liu, LLMs: Understanding code syntax and semantics for code analysis, arXiv preprint arXiv:2305.12138 (2023).
- [33] E. V. de Paulo Sobrinho, A. De Lucia, M. de Almeida Maia, A systematic literature review on bad smells–5 w's: which, when, what, who, where, IEEE Transactions on Software Engineering, 47 (1) pp.17–66, (2018)
- [34] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software, 138 , pp.158–173, (2018)
- [35] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proc. of the 20th int'l conf. on evaluation and assessment in software engineering, 2016, pp. 1–12.
- [36] M. S. Haque, J. Carver, T. Atkison, Causes, impacts, and detection approaches of code smell: a survey, in: Proc. of the 2018 ACM Southeast Conference, 2018, pp. 1–8.
- [37] R. S. Menshawy, A. H. Yousef, A. Salem, Code smells and detection techniques: a survey, in: Proc. of 2021 int'l mobile, intelligent, and ubiquitous computing conference (MIUCC 2021), IEEE, 2021, pp. 78–83.
- [38] A. Al-Shaaby, H. Aljamaan, M. Alshayeb, Bad smell detection using machine learning techniques: a systematic literature review, Arabian Journal for Science and Engineering, 45 (4) pp.2341–2369, (2020)
- [39] A. Alazba, H. Aljamaan, M. Alshayeb, Deep learning approaches for bad smell detection: a systematic literature review, Empirical Software Engineering, 28 (3), 77, (2023)
- [40] F. Pecorelli, F. Palomba, D. D. Nucci, A. D. Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, in: Proc. of the 2019 IEEE/ACM 27th Int'l Conf. on Program Comprehension (ICPC 2019), Montreal, Canada, 2019, pp. 1–11, IEEE.
- [41] D. Mesbah, N. E. Madhoun, K. A. Agha, H. Chalouati, Leveraging prompt-based large language models for code smell detection: A comparative study on the mlcq dataset, in: Proc. of The 13th Int'l Conf. on Emerging Internet, Data & Web Technologies (EIDWT-2025), Springer, Matsue, Japan, 2025, pp. 444–454.
- [42] PMD, PMD: A source code analyzer, <https://pmd.github.io/>, accessed: 2025-02-26.
- [43] Checkstyle, Checkstyle: A development tool to help you write Java code that follows a coding standard, <https://checkstyle.sourceforge.io/>, accessed: 2025-02-26.
- [44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, et al., Experimentation in software engineering, Springer, 2012.
- [45] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., CodeBert: A pre-trained model for programming and natural languages, arXiv preprint arXiv:2002.08155 (2020).
- [46] Y. Wang, W. Wang, S. Joty, S. C. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859 (2021).
- [47] R. Jonnala, J. Yang, Y. Lee, G. Liang, Z. Cao, Measuring and improving the efficiency of Python code generated by LLMs using CoT prompting and fine-tuning, IEEE Access (2025).