Master's Thesis

Code Quality of Open-Source LLM: A Code Smell Analysis

Qingqing Dong

December 2, 2025

Advisor:
Dr. Norman Peitek Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Jilles Vreeken CISPA Helmholtz Center for Information Security

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich zur Textüberarbeitung/-korrektur und zur Code-Vervollständigung und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Ich erkläre mich damit einverstanden, dass die Arbeit mittels eines Plagiatsprogrammes auf die Nutzung einer solchen Software überprüft wird. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

I hereby declare that I have written this thesis independently and without the involvement of third parties, and that I have used no sources or aids other than those indicated. All passages taken directly or indirectly from publications or other external sources have been identified as such. In particular, I confirm that I have used AI-based software (e.g., ChatGPT) exclusively for the following permitted sub-tasks: text rewriting/revision and code completion , and not to address or formulate the main research questions of the thesis. All parts of the thesis that were generated and/or edited using AI-based software (e.g., ChatGPT) have been disclosed and documented in accordance with the documentation requirements. I agree that the thesis may be checked using plagiarism detection software, including checks for the use of such software. I am aware that any violation of this declaration may result in failing the examination and lead to losing the right to be examined.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Einverständniserklärung (optional) Declaration of Consent (optional)

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Abstract

Software engineering workflows increasingly incorporate Large Language Models (LLMs), yet most evaluation metrics remain centered on functional correctness. In contrast, the internal quality and long-term maintainability of Artificial Intelligence (AI)-generated code—assessed through code smells—are still largely unexplored. This thesis addresses two key questions: (1) How do the quality profiles of open-source LLMs compare to those of human developers? Moreover, (2) Can prompt engineering meaningfully improve code quality? We conduct a systematic study of three open-source LLMs (Phi-3-mini, Phi-4, Qwen2.5-Coder) across two realistic benchmarks (CoderEval and BigCodeBench-Hard). Our analysis spans 6,048 code samples, including 5,670 LLM-generated solutions produced with five prompting strategies (Zero-Shot, Quality-Focused, Persona-Based, Chain of Thought (CoT), Recursive Criticism and Improvement (RCI)) and 378 human-written canonical solutions, evaluated using industry-standard static analysis tools (Pylint, Bandit).

Our findings reveal three central insights. First, LLMs exhibit a pronounced Syntax–Logic Gap: although 98.5% of generated code is syntactically valid, 52% – 78% contain at least one code smell, with *E0602: Undefined variable* among the most common, reflecting both genuine hallucinations and the models' reliance on implicit project contexts. Second, while LLMs produce fewer total smells than humans on average (1.77 vs. 4.27), their quality profiles differ substantially—human code tends toward stylistic violations. In contrast, LLM-generated code frequently contains structural errors such as undefined variables and namespace collisions. Third, simple constraint-based prompts reduce smell density by 7% – 15%, but more complex recursive self-correction strategies consistently degrade quality through verbosity and hallucinated refactoring.

Overall, this thesis provides a systematic analysis of code smells in open-source LLMs, establishing internal code quality as a measurable and essential dimension of AI-assisted software engineering.

Contents

List of Figures

List of Tables

Listings

Acronyms

LLM	Large Language Model
AI	Artificial Intelligence
IDE	Integrated Development Environment
CoT	Chain of Thought
RCI	Recursive Criticism and Improvement
RLHF	Reinforcement Learning from Human Feedback
MBPP	Mostly Basic Python Problems
AST	Abstract Syntax Tree
IQR	Interquartile Range
ANOVA	Analysis of Variance
API	Application Programming Interface
RAG	Retrieval-Augmented Generation

1 Introduction

In recent years, Generative AI has rapidly become an integral part of everyday digital workflows. It assists with tasks ranging from drafting emails and answering questions to writing code and debugging software. Among the most widely used Generative AI systems is ChatGPT, which, since its public release in late 2022, has popularized the capabilities of LLMs and brought them into mainstream awareness. At their core, LLMs are deep neural networks designed to predict and generate coherent natural language.

The integration of LLMs into software engineering workflows predates the launch of ChatGPT. Even before 2022, tools such as GitHub Copilot [24] had already begun reshaping how developers approach coding tasks. GitHub Copilot assists by offering autocomplete-style code suggestions and full-function generation within Integrated Development Environments (IDEs). According to a 2022 user research by GitHub, 88% of developers report being more productive, and 96% state that Copilot helps them complete repetitive tasks faster [23].

These tools transform the nature of software development itself. From automated testing to code refactoring, deployment scripts, and even architecture suggestions, modern LLMs can automate progressively complex phases of the development lifecycle. According to the 2024 Stack Overflow Developer Survey [57], over 62% of professional developers already use AI-based tools in their workflows, with usage growing rapidly across all experience levels.

However, alongside this rapid adoption and substantial productivity gains, a critical question remains: What is the quality of AI-generated code? Prior research has evaluated LLM-generated code primarily on syntactic correctness, execution accuracy, and security vulnerabilities [48, 82]. While these criteria are important, they capture only part of what constitutes high-quality code. Code quality also encompasses readability, maintainability, and long-term understandability—dimensions that remain absent from functional or security-focused metrics.

For example, a code snippet may pass unit tests and appear secure, yet still suffer from poor structure and low maintainability. We often capture these quality issues through code smells—indicators in code that signal deeper structural problems such as low cohesion or high coupling Fowler et al. [20]. Code smells might cause delayed failures rather than immediate runtime errors, yet they have been linked to increased fault-proneness and reduced maintainability [19, 85]. Recent studies have shown that code generated from GitHub Copilot and other LLMs often contains multiple smells per snippet [69, 70].

Another underexplored dimension is the role of prompt engineering. Research has shown [65] that techniques like Zero-Shot, Few-Shot, and CoT reasoning can significantly affect the output of LLMs on various tasks. More advanced approaches, such as persona-based priming and refinement-based techniques like RCI, further expand how prompts can guide model behavior. One study demonstrated that different prompts significantly

affect the security level of code generated by GPT-4 [78]. However, how these techniques affect code quality—particularly the prevalence and types of code smells—remains largely unstudied.

Despite growing research on LLM code generation, three critical gaps remain: (1) lack of systematic evaluation of code smells in open-source models using realistic benchmarks, (2) absence of comparative analysis against human-written baselines, and (3) unexplored impact of prompt engineering on internal code quality.

To address this research gap, this thesis investigates the intersection of LLM-generated code, code smells, and prompt engineering, with a particular focus on open-source LLMs and realistic development tasks. Specifically, we evaluate three leading open-source models: Phi-4, Phi-3-mini-128k-instruct, and Qwen2.5-Coder-32B-Instruct. While prior research has explored LLM performance in terms of correctness and security, few studies have systematically evaluated the prevalence and types of code smells introduced during generation—especially under different prompting strategies. We employ industry-standard static analysis tools (Pylint and Bandit) to systematically detect and categorize code smells across thousands of generated samples, providing both maintainability and security-focused assessments. We evaluate models on modern benchmarks that better represent the complexity of real-world software development. To guide this investigation, we establish the following objectives:

1. **Systematic Code Smell Assessment**: Conduct a thorough evaluation of code smells in code generated by leading open-source LLMs, measuring how often different types of smells appear and their distributions.

2. **Real-World Benchmark Evaluation**: Test model performance on realistic coding benchmarks, specifically CoderEval and BigCodeBench-Hard, which better represent the complexity of real-world software development compared to earlier, synthetic benchmarks like HumanEval or Mostly Basic Python Problems (MBPP).

3. **Prompt Engineering Impact Analysis**: Analyze how different prompt engineering techniques—including Zero-Shot, Quality-Focused, Persona-Based, CoT, and RCI—affect the number, types, and distribution of code smells in generated code.

To achieve these objectives, this thesis addresses the following research questions: **RQ1: How prevalent are code smells in code generated by leading open-source LLMs?**

- **RQ1.1**: What is the frequency and distribution of different code smell types in LLM-generated code?

- **RQ1.2**: How does the prevalence of smells in LLM-generated code compare to that in human-written canonical solutions?

RQ2: Can strategic prompt engineering prevent code smells in LLM-generated code?

- **RQ2.1**: How do different prompt engineering techniques affect the total count of generated code smells?

- **RQ2.2**: Do specific prompting techniques influence the types of code smells generated?

1.1 Thesis Organization

We organize the rest of this thesis as follows: Chapter 2 reviews key concepts about LLMs, code generation, and common code smells. It also explains current prompt engineering techniques and how they influence software development. Chapter 3 defines the research questions in detail, describes the experimental setup, models, and benchmarks used, and explains the method for evaluating code smells. Chapter 4 presents the results from our experiments, including comparisons across different models and prompting strategies. Chapter 5 presents qualitative results that illustrate key findings. We also discuss potential limitations and threats to validity. Chapter 6 reviews related research on evaluating the quality of LLM-generated code and positions our work within this broader context. Chapter 7 summarizes our main contributions and suggests directions for future research in LLM-assisted software engineering.

2

Background

This chapter establishes the theoretical and technical foundations of this thesis. Understanding these foundations is essential for interpreting our empirical findings on code smell prevalence and the efficacy of prompt engineering interventions. We begin by exploring the architecture, capabilities, and inherent limitations of LLMs in software engineering. Subsequently, we discuss the characteristics and advantages of open-source models. We then examine prompt engineering techniques and review the benchmarks used to evaluate code generation. Finally, we define code smells and discuss the static analysis tools employed to detect them.

2.1 Large Language Models

LLMs are deep neural networks trained to generate coherent text by predicting the next token in a sequence, based on the surrounding context. Most modern LLMs are based on the Transformer architecture [80], which enables efficient learning over large datasets with long-range dependencies through its self-attention mechanism. Training LLMs typically involves two distinct stages: pretraining and fine-tuning. In the pretraining phase, the model learns general language and code patterns from large corpora using self-supervised tasks such as next-token prediction. For programming-specific models, training datasets often include large code corpora such as CodeSearchNet [32], The Pile [22], and GitHub repositories. Fine-tuning then adapts the model to specific domains or tasks—often using supervised learning or Reinforcement Learning from Human Feedback (RLHF) to better align with human intent and preferences [56].

At inference time, LLMs function by sampling one token at a time using various decoding strategies (e.g., greedy decoding, nucleus sampling) until reaching a completion condition. Model performance generally improves with scale, following scaling laws that describe the relationship between model size, training data, and computational resources [36]. These scaling laws suggest that larger models trained on more data achieve better performance on downstream tasks. However, larger models also require significantly more memory and computational power, creating a trade-off between performance and efficiency.

2.1.1 Applications and Capabilities

Several state-of-the-art LLMs have demonstrated exceptional capabilities across diverse domains, including complex reasoning and advanced code generation [10, 15, 47]. These

models now exhibit human-level or superior performance in highly specialized fields. For example, GPT-4 passed the Uniform Bar Exam with a score in the 90th percentile [37] and achieved 90.4% accuracy on clinical vignette-based questions from the USMLE test, significantly surpassing the medical student average of 59.3% [9]. In complex scientific reasoning, AI models have solved grand challenges, such as DeepMind's AlphaFold achieving unprecedented accuracy in protein structure prediction—a breakthrough recognized with the 2024 Nobel Prize in Chemistry [35].

In software engineering, we observe this performance most visibly in competitive programming benchmarks. DeepMind's AlphaCode achieved a significant milestone, reaching an average ranking in the top 54.3% of participants in competitions with over 5,000 competitors [47]. More recently, a 2025 study on the CodeElo benchmark found that OpenAI's o1-mini model achieved a rating surpassing nearly 90% of human participants [63].

This capability extends beyond contests to core, traditionally manual software engineering tasks. In automated bug detection and repair, models demonstrate high efficacy. A 2025 study using GPT-4 on the Defects4J dataset achieved bug-detection accuracy of 89.7% and mitigation efficacy of 86.4% [64]. AI agents like Google's CodeMender are actively being deployed, contributing over 70 security fixes to large-scale open-source projects [61]. Hybrid approaches, such as CODAMOSA, integrate LLMs with traditional search-based software testing to significantly improve test coverage [44]. Furthermore, in code refactoring, recent studies on Extract Method refactoring found that models could correctly preserve functionality while improving code metrics, achieving over 70% developer acceptance [11].

2.1.2 Limitations and Risks

Despite these advancements, LLMs possess fundamental limitations that pose risks to software quality. The most widely recognized limitation is the tendency to hallucinate—generating outputs that are fluent and plausible-sounding, yet factually incorrect or ungrounded [33]. Hallucination arises because LLMs are fundamentally probabilistic systems optimized for sequence prediction rather than factual verification [90]. Furthermore, their reasoning is brittle; models struggle with complex, multi-step logical problems, and performance degrades significantly when we alter the problem structure [13]. Knowledge cutoffs also confine a model's capabilities to its static training data, rendering it unaware of recent libraries or updates [45]. Finally, LLMs inherit and can amplify ingrained biases from their training corpora, potentially reinforcing harmful societal stereotypes [21].

In high-stakes domains, these general limitations escalate to severe practical consequences. In medicine, model-generated hallucinations become direct threats to patient safety, carrying the potential for misdiagnosis and inappropriate treatment recommendations [6]. This translation of general probabilistic error into acute, domain-specific risk poses an equally significant challenge in software engineering, where these fundamental limitations manifest as acute risks to security, reliability, and long-term maintainability [1, 18, 81].

A primary risk is the generation of insecure code; a 2025 analysis found that up to 45% of AI-generated programs carried exploitable bugs [81]. This risk extends to the software supply chain through package hallucination, in which models invent nonexistent libraries.

A 2025 USENIX Security study found that open-source models hallucinated packages in 21.7% of cases, creating vectors for malicious injection [43, 74].

Even when generated code is functionally correct and secure, it often suffers from significant non-functional quality issues [1]. This reveals a critical mismatch: while academic benchmarks focus on functional correctness, industry experts prioritize maintainability, warning that AI-generated code may accelerate technical debt [71]. Empirical studies confirm that LLMs tend to produce higher cyclomatic complexity [38]. In complex problems, LLM-generated solutions may introduce structural issues absent in human-written code, such as logic inefficiencies co-occurring with poor readability [1, 75]. These failures stem from the models' reliance on pattern matching rather than a proper understanding of program architecture [27].

To effectively balance productivity gains against these risks, we must define performance beyond simple correctness. This thesis addresses this gap by focusing on a critical, yet underexplored, metric: internal code quality as indicated by code smells.

2.1.3 Open-Source Models

We can categorize LLMs as closed-source (e.g., GPT-4, Gemini) or open-source (e.g., DeepSeek, Llama). Open-source models offer distinct advantages for academic research: full accessibility without Application Programming Interface (API) costs or rate limits, transparency in model architecture enabling detailed analysis, and reproducibility of experimental results [52]. These characteristics are particularly valuable for studies requiring extensive experimentation, such as code quality evaluation across thousands of generated samples.

Open-source models have gained significant traction in recent years, increasingly serving as the foundation for domain-specific adaptations in software engineering. Major collaborative initiatives demonstrate this trend: the BigScience BLOOM project [4], coordinated by hundreds of researchers worldwide. At the same time, the BigCode effort produced StarCoder [46], a model trained explicitly on source code from permissively licensed repositories. Parameter-efficient fine-tuning techniques like LoRA [28] further accelerate specialization, enabling researchers to adapt pre-trained models to specific tasks with limited computational resources.

The emergence of standardized benchmarks and leaderboards, such as BigCodeBench [91], provides frameworks for comparing open-source code generation models across diverse programming tasks. These evaluation platforms track performance metrics and facilitate model selection for both research and practical applications. The specific open-source models employed in this thesis, along with the rationale for their selection, are detailed in Chapter 3.

2.2 Prompt Engineering Techniques

Since LLMs are autoregressive models, they are susceptible to input phrasing, structure, and formatting. Even minor modifications can significantly alter output, highlighting the need to understand how prompts affect generated code [67]. Prompt engineering refers

to the deliberate design of inputs to guide and optimize model behavior [49]. Researchers have developed various techniques to improve reliability and code quality, which we can categorize by their operational approach:

Pattern-Based Techniques These techniques rely on providing examples to establish output patterns.

- **Zero-Shot** [10]: The model receives a task description without examples (e.g., *Write a Python function to reverse a string*). While efficient, it may fail to elicit an optimal code structure.
- **One-Shot** and **Few-shot** [10]: These approaches provide one or multiple input–output examples. This establishes a template for the model to replicate, which is particularly effective for enforcing coding conventions and has demonstrated substantial performance improvements across various tasks [10].

Role-Based Techniques

- **Persona-based**: We instruct the model to adopt a specific role, such as *You are a senior software engineer*. This can influence the tone, technical depth, and quality standards of the generated output [42].

Reasoning-Based Techniques These techniques encourage the model to articulate its thought process.

- CoT: We incorporate phrases like *Let's think step by step* to elicit intermediate reasoning. [84] showed that CoT improves performance in symbolic reasoning and enhances interpretability.
- **Self-planning** [34]: We decompose a task into a structured plan before implementation. This guides models to generate more logically structured and modular code.

Self-Improvement Techniques These techniques involve iterative refinement.

- RCI: A multi-stage process where the model generates a solution, critiques it, and refines it based on self-assessment [41]. This approach significantly improves reasoning performance across natural language benchmarks, surpassing CoT supplemented with external feedback.
- **Self-refine** [51]: Structured around generation, feedback, and refinement phases using few-shot demonstrations of <*input*, *output*, *feedback*> triplets.

2.3 Code Generation

Code generation involves automatically synthesizing source code from natural-language descriptions. Typical tasks include **code completion**, **test case synthesis**, and **function generation**. Function generation encompasses creating complete implementations from

descriptions or docstrings. While developers often interact iteratively with LLMs, studies show that models can generate correct code from a single prompt [12]. We adopt a function-generation framework in this thesis because it enables the evaluation of inherent programming abilities and code quality without the confounding factors introduced by conversational interactions.

2.3.1 Evaluation Benchmarks

Standardized benchmarks are essential for measuring model capabilities. HumanEval [12] and MBPP [8] are foundational benchmarks comprising concise, self-contained Python problems (164 and 974 tasks, respectively) focused on algorithmic reasoning and standard library usage. However, these traditional benchmarks contain short, algorithm-focused tasks that may fail to capture the complexity of real-world development. Additionally, their age raises concerns about data contamination, as models may have memorized these problems during training [31].

To address these limitations and evaluate code quality in realistic scenarios, modern benchmarks have been developed that better reflect industrial software engineering practices:

CoderEval [86] A framework grounded in real-world practices, comprising 230 Python functions extracted from popular GitHub projects. Unlike synthetic benchmarks, CoderEval tasks include contextual dependencies that require understanding of broader code structure and library interactions. Each task presents a function signature with a docstring describing the required functionality, mirroring actual development scenarios where developers implement functions within existing code bases.

BigCodeBench-Hard [91] A curated subset of the complete BigCodeBench benchmark, containing 148 of its most challenging tasks spanning domains like web development, data analysis, and systems programming. This subset emphasizes multi-step problem-solving and integration with external libraries. Tasks are selected based on three criteria: requiring more than two external libraries, having ground-truth solutions longer than 426 tokens (the benchmark average), and achieving solve rates below 50% across previously evaluated models. This selection process ensures that the benchmark tests genuine programming ability rather than pattern matching from training data.

2.4 Code Smells

Code smells are surface-level indicators of potential problems in software design, maintainability, or readability. Originally introduced by Fowler et al. [20], they signal deeper structural issues—such as low cohesion or high coupling—that do not necessarily cause immediate functional failures. However, over time, these issues can progressively degrade maintainability [85], reduce extensibility, and increase fault-proneness [40].

While the principle of detecting poor design is universal, the specific taxonomy of smells varies by programming language. In Python, common code smells include:

1. **Readability issues**: Violations of style guidelines like PEP 8 (e.g., excessive line length) [25].
2. **Structural inefficiencies**: Unused variables, imports, or dead code indicating poor planning.
3. **Design anomalies**: Classes with too few public methods or unnecessary object-oriented complexity.
4. **Encapsulation violations**: Inappropriate access to protected members.

In the context of LLM-generated code, the presence of these smells may indicate shallow reasoning patterns or an inadequate understanding of software engineering best practices.

2.4.1 Static Analysis Tools

To identify code smells systematically at scale, we rely on automated static analysis tools in this thesis. Static analysis is particularly well-suited for large-scale LLM evaluation because it enables the consistent, reproducible detection of quality issues across thousands of samples without execution. The tool landscape is diverse; within Java, tools like JDeodorant [79] and PMD [58] are standard. In Python, the ecosystem includes tools such as Radon [66] for code complexity and Pysmell [14]. However, accuracy varies between tools, underscoring the importance of selecting detectors that align with specific study goals [29].

We employ **Pylint** [50] and **Bandit** [17] to analyze LLM-generated Python programs. We selected these tools because they provide a complementary assessment of code quality [69]. Pylint evaluates maintainability and readability, while Bandit addresses security concerns. Together, they form a comprehensive view of internal code quality.

Pylint Pylint is a comprehensive static analyzer that categorizes issues into classes ranging from stylistic deviations to critical errors [76]. Table 2.1 provides a description and example for each category, illustrating the range of issues detected from stylistic conventions to critical errors. In addition, Pylint generates detailed diagnostic reports and assigns numeric quality scores, providing a quantitative framework for assessing code quality.

Bandit Bandit specializes in identifying security-related smells. It detects patterns such as unsafe functions (e.g., `eval`), hardcoded credentials, and insecure random number generation. These security smells are critical for assessing the reliability of production-level systems.

Static analysis tools have inherent limitations that we must acknowledge. Code smell definitions can be subjective, and tools may produce false positives or fail to detect semantically complex smells [60]. Furthermore, heuristic-based detection relies on rigid metric thresholds that may fail to align with specific project contexts [73]. Consequently, we should view tool outputs as probabilistic indicators of structural quality rather than absolute measures of correctness. Despite these limitations, the combination of Pylint and Bandit provides a robust, holistic methodology for evaluating the internal quality of code generated by LLMs.

Table 2.1: Pylint Message Categories (Convention, Refactor, Warning, Error) with Descriptions and Code Examples Illustrating Each Violation Type

Category	Description	Example Violation

Convention	Violations of coding standards (e.g., PEP 8) and missing docstrings.	**Missing function docstring (C0116):**
		`def print_python_version():`
		`  print(sys.version)`

Refactor	Structural inefficiencies or code smells, such as duplicated blocks.	**No else return (R1705):**
		`if a == b:`
		`  return 0`
		`elif a < b:`
		`  return -1`
		`else:`
		`  return 1`

Warning	Stylistic problems or minor issues that may lead to bugs in edge cases.	**Unused variable (W0612):**
		`def print_fruits():`
		`  fruit1 = "orange"`
		`  fruit2 = "apple"`
		`  print(fruit1)`

Error	Critical issues representing likely bugs or syntax errors.	**Undefined variable (E0602):**
		`def print_value():`
		`  print(number + 2)`

3

Methodology

This chapter details the empirical methodology we designed to investigate the quality of code generated by modern, open-source LLMs. We present the guiding research questions, the experimental framework, the materials used, and the procedures we followed for data collection and analysis.

3.1 Research Questions

As established in Chapter 1, this study addresses two primary research questions, each with targeted sub-questions:

RQ1 How prevalent are code smells in code generated by leading open-source LLMs?

- **RQ1.1**: What is the frequency and distribution of different code smell types in LLM-generated code?
- **RQ1.2**: How does the prevalence of smells in LLM-generated code compare to that in human-written canonical solutions?

RQ2 Can strategic prompt engineering prevent code smells in LLM-generated code?

- **RQ2.1**: How do different prompt engineering techniques affect the total count of generated code smells?
- **RQ2.2**: Do specific prompting techniques influence the types of code smells generated?

We address these questions through complementary experiments as detailed in Section 3.3.2. Experiment 1 establishes baseline smell prevalence across models and compares against human-written code, directly addressing RQ1. Experiment 2 evaluates the efficacy of five prompt engineering techniques on smell reduction and distribution, addressing RQ2.

3.2 Experimental Setup

This section details and justifies each component of the experimental setup. As Chapter 2 introduced the general concepts of models, benchmarks, and tools, this section focuses on the rationale for their selection and their specific roles in the experimental design.

3.2.1 Model Selection

As discussed in Section 2.1.3, this thesis focuses exclusively on open-source LLMs to ensure transparency, reproducibility, and accessibility. We selected three models based on advanced programming proficiency, computational efficiency for local deployment, and strong community adoption. Importantly, all three rank among the top-performing open-source models on the BigCodeBench Leaderboard [91].

The selected models, summarized in Table 3.1, represent a strategic distribution of parameter sizes (3.8B, 14B, and 32B). This range allows for investigation into how model scale influences the generation of maintainable, low-smell code.

Table 3.1: Characteristics of Selected Open-Source Large Language Models (Phi-3-mini, Phi-4, Qwen2.5-Coder) Including Developer and Parameter Count

Model Name	**Developer**	**Parameter Count**
Phi-3-mini-128k-instruct	Microsoft	3.8 billion
Phi-4	Microsoft	14 billion
Qwen2.5-Coder-32B-Instruct	Alibaba Cloud	32 billion

The specific characteristics of each model are as follows:

Phi-3-mini-128k-instruct [2] A compact 3.8-billion parameter model designed for computational efficiency. The 128k token context window enables the processing of lengthy code contexts, making it suitable for tasks that require an understanding of the broader project structure. Despite its small size, Phi-3-mini demonstrates competitive performance on coding benchmarks, making it suitable for resource-constrained environments.

Phi-4 [3] A 14-billion parameter general-purpose model developed by Microsoft. Phi-4 exhibits advanced reasoning capabilities and has undergone rigorous alignment procedures to ensure safe, helpful outputs. Despite its moderate size, it performs competitively on complex coding tasks.

Qwen2.5-Coder-32B-Instruct [77] An advanced code-specialized LLM developed by Alibaba Cloud. Trained on extensive multi-language repositories totaling over 5.5 trillion tokens, it is specifically optimized for high-quality code generation across multiple programming languages and paradigms. This model represents the current frontier of open-source code generation capabilities.

All models were deployed locally using the *Hugging Face Transformers* library with consistent inference parameters (`temperature=0.7`, `max_new_tokens=512`) to ensure fair comparison across platforms. Throughout the Results and Discussion chapters, we adopt abbreviated identifiers for readability: *Phi-3-mini* refers to Phi-3-mini-128k-instruct, and *Qwen2.5-Coder* refers to Qwen2.5-Coder-32B-Instruct.

3.2.2 Benchmark Selection

To ensure the evaluation reflects authentic software engineering challenges, we utilize two modern benchmarks: CoderEval [86] and BigCodeBench-Hard [91]. As detailed in Section 2.3.1, these benchmarks move beyond the simple, self-contained algorithmic problems typical of older benchmarks and are less susceptible to data contamination. For the experiments conducted in this thesis, we utilize the complete sets of both the CoderEval (Python subset) and BigCodeBench-Hard benchmarks. We provide an overview in Table 3.2.

Table 3.2: Comparison of Evaluation Benchmarks (CoderEval and BigCodeBench-Hard) by Task Count and Primary Evaluation Focus

Benchmark Name	Number of Python Tasks	Core Focus
CoderEval	230	Contextual Dependency
BigCodeBench-Hard	148	Task Complexity & Tool Use

We use CoderEval [86] to assess the ability of a model to generate code that functions within an existing, complex code base. A typical task presents the model with a function signature accompanied by a docstring describing the required functionality, as shown in Listing 3.1. We then expect the model to generate the body of the function.

Listing 3.1: Example Task of CoderEval Benchmark

```
def hydrate_time(nanoseconds, tz=None):
    """
    Convert nanoseconds to a time in fixed format.
    """
```

In contrast, we use BigCodeBench-Hard [91] to evaluate a model's capacity to solve complex, multi-step problems that require sophisticated reasoning and the use of multiple external libraries. A representative example appears in Listing 3.2.

3.2.3 Prompt Design

To investigate RQ2, we selected five distinct prompt engineering techniques. These techniques, detailed in Table 3.3, are widely studied and represent a spectrum of approaches, from simple directives to complex, multi-step reasoning frameworks.

Zero-Shot is a baseline approach with no examples or explicit guidance. **Quality-Focused** is a variant of Zero-Shot prompting that includes explicit constraints instructing the model to prioritize clean code. **Persona-Based** prompting instructs the model to adopt the persona of a software quality expert [42]. **CoT** prompting encourages the model to articulate its reasoning process before generating code. **RCI** is a multi-step technique where the model generates, critiques, and refines its own solution.

Listing 3.2: Example Task of BigCodeBench-Hard

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

def task_func(data, n_components=2):
    """
    Perform Principal Component Analysis (PCA) on a dataset and record the result.
    Also, generates a scatter plot of the transformed data.

    Parameters:
    data (DataFrame): The dataset.
    n_components (int): The number of principal components to calculate. Default is 2.

    Returns:
    DataFrame: The transformed data with principal components.
    Axes: The matplotlib Axes object containing the scatter plot.

    Raises:
    ValueError: If n_components is not a positive integer.

    Requirements:
    - numpy
    - pandas
    - matplotlib.pyplot
    - sklearn.decomposition

    Example:
    >>> data = pd.DataFrame([[14, 25], [1, 22], [7, 8]], columns=['Column1', 'Column2
        '])
    >>> transformed_data, plot = task_func(data)
    """
```

We deliberately included a standard instruction, *Avoid docstrings and comments*, in all prompt templates. Preliminary experiments showed that models often generate verbose docstrings. This behavior frequently caused the generated output to exceed the maximum token limits, resulting in incomplete code and significantly slower generation times. By explicitly instructing the models to omit them, we ensure that the analysis remains focused on the quality of the functional code itself.

3.2.4 Analysis Tools

We used automated static analysis to objectively and reproducibly identify code smells. We chose the tools Pylint and Bandit, summarized in Table 3.4, due to their comprehensive rule sets and widespread adoption in both industry and academic research [55, 69, 70].

Table 3.3: Prompt Engineering Techniques (Zero-Shot, Quality-Focused, Persona-Based, CoT, RCI) with Complete Template Specifications

Technique Name	Prompt Template
Zero-Shot	Generate Python code for the following task. *Avoid docstrings and comments.* **<task>**
Quality-Focused	Generate Python code for the following task, ensuring it is clean and avoids code smells. *Avoid docstrings and comments.* **<task>**
Persona-Based	Act as a software quality expert. Provide outputs that a quality expert would give. Generate Python code for the following task. *Avoid docstrings and comments.* **<task>**
CoT	Step 1: Generate Python code for the following task, ensuring it is clean and avoids code smells. *Avoid docstrings and comments.* Let's think step by step. **<task>** Step 2: Therefore, final Python implementation without docstrings or comments is:
RCI	Step 1: Generate Python code for the following task. *Avoid docstrings and comments.* **<task>** Step 2: Review your previous answer and find code smells with it. Step 3: Improve your code based on the code smells you found. Therefore, final Python implementation without docstrings or comments is:

Table 3.4: Static Analysis Tools (Pylint and Bandit) with Primary Purpose and Categories of Code Smells Detected

Tool Name	Primary Purpose	Key Issue Categories Detected
Pylint	General Code Quality & Maintainability	Error, Warning, Refactor, Convention
Bandit	Security Vulnerability Detection	Security Smells

A key methodological decision in this study is the definition of a *code smell*. As there is no single, formal definition, this thesis follows the precedent of prior research by using the term as a broad umbrella for any issue flagged by our static analysis tools that indicates a potential problem with the code's quality, maintainability, or correctness. This includes not only stylistic *Convention* violations or *Refactor* candidates but also Pylint *Error* category messages, such as *E0602: Undefined variable*. We treat them here as the most severe type of code smell detectable through static analysis.

We conducted all analyses using Pylint 3.3.7 and Bandit 1.8.6 with their default parameter settings to ensure a standardized and replicable baseline. To focus the analysis on substantive issues, we excluded a predefined set of stylistic and context-dependent messages from Pylint. We detail these exclusions in Section 3.3.1.

3.3 Experimental Procedure

This section describes the execution of the experiments and the methods we used to process and analyze the collected data. We conducted a substantial empirical investigation. For each of the three selected models and five prompts, we generated solutions for all 230 tasks in CoderEval and 148 tasks in BigCodeBench-Hard. This process generated 5,670 unique code solutions. In addition, we also analyzed the 378 canonical human-written solutions from both benchmarks, bringing the total number of evaluated code samples to over 6,000.

As depicted in Figure 3.1, we designed a multi-stage experimental pipeline to address the research questions systematically. The process begins by selecting a programming task from a benchmark dataset. We then apply a prompt engineering strategy to formulate an input for a given LLM. The model generates a code solution, which then enters an analysis pipeline where we evaluate it for syntactic correctness and code smells.

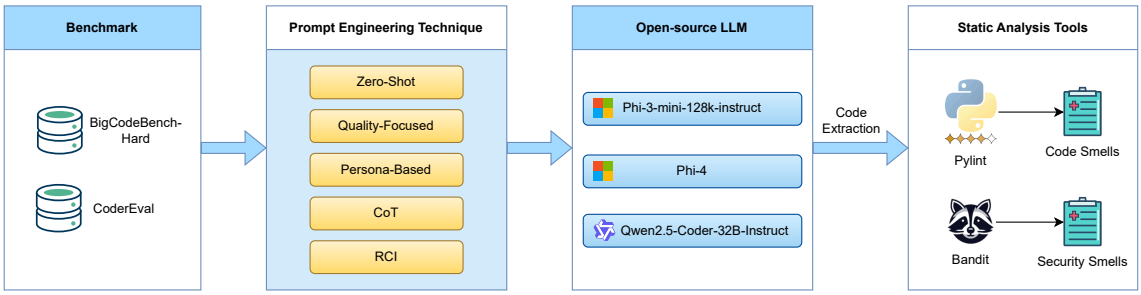

Figure 3.1: Multi-Stage Experimental Pipeline for Code Generation, Static Analysis, and Code Smell Detection Across Models, Benchmarks, and Prompting Techniques

3.3.1 Data Preprocessing

For each combination of LLM, benchmark task, and prompt engineering technique, we generated a single code solution. The generated output then underwent a three-step preprocessing phase.

1. **Code Extraction**: The process of isolating executable code from the raw output of the model involved two stages. First, we extracted the relevant code snippet from a structured JSON response. The specific field depended on the prompting technique; for multi-step methods like RCI and CoT, we selected the final refined solution. For single-step prompts, we used direct generation. Second, since this extracted text could still contain explanatory language, we applied a heuristic parser. This parser identified the start of the code block by searching for common Python keywords (e.g., `def`, `import`) and the end by detecting natural language phrases (e.g., *example usage:*, *this function will*) that signal the conclusion of the code. This two-stage process ensured that we retained only the intended executable code for analysis.

2. **Syntax Filtering**: We parsed each extracted code snippet using Abstract Syntax Tree (AST) We retained only syntactically valid solutions for analysis, as static analysis tools fail to operate on code that fails to parse.

3. **Smell Filtering**:To focus the analysis on substantive structural and logical issues, we filtered the Pylint results by excluding messages that are irrelevant to the core quality of the generated code. This methodology, consistent with related work [69, 70], removes noise from four main categories. First, we ignored stylistic messages, such as whitespace or naming conventions, because they do not significantly affect program logic or maintainability. Second, we excluded *Warnings* for missing documentation because our experimental prompts explicitly instructed the models to omit docstrings. Third, we filtered out import-related errors, as import analysis of isolated function-level code snippets is unreliable without full project context. Finally, we disregarded *Fatal* errors, as they indicate failures of the analysis tool itself rather than problems in the source code. Table 3.5 provides a comprehensive list of all excluded Pylint messages and their corresponding categories.

3.3.2 Experimental Design

To address the research questions, we conducted experiments to provide a comprehensive evaluation of code smells in code generated by open-source LLMs.

Experiment 1 To address RQ1, we assessed the frequency of code smells under standard conditions. For each LLM and benchmark, we generated solutions for all tasks using the **Zero-Shot** prompt, and we analyzed each syntactically correct solution with Pylint and Bandit. To answer RQ1.1 (Frequency and Distribution), we collected the smell data from all generated solutions to establish a baseline of code smell prevalence. To answer RQ1.2 (Comparison with Canonical Answers), we subjected the human-written canonical solutions from the benchmarks to the same analysis procedure to ensure a fair comparison between the LLM-generated code and the canonical solutions.

Experiment 2 To address RQ2, we repeated the process of code generation and analysis from Experiment 1 using all five prompt engineering techniques. This enabled a direct comparison of the effects of each prompt on the number and types of code smells produced.

Table 3.5: Pylint Messages Excluded from Analysis Organized by Category

Category	Type	Message
Style and Naming	Convention	C0303: Trailing whitespace
	Convention	C0304: Missing final newline
	Convention	C0305: Trailing newlines
	Convention	C0103: Invalid name
Missing Documentation	Convention	C0112: Empty docstring
	Convention	C0114: Missing module docstring
	Convention	C0115: Missing class docstring
	Convention	C0116: Missing function docstring
Imports	Warning	W0611: Unused import
	Warning	W0401: Wildcard import
	Refactor	R0402: Consider using from-import
	Convention	C2403: Non-ASCII module import
	Warning	W0404: Reimported
	Warning	W0614: Unused wildcard import
	Convention	C0410: Multiple imports
	Convention	C0411: Wrong import order
	Convention	C0412: Ungrouped imports
	Convention	C0413: Wrong import position
	Convention	C0414: Useless import alias
	Convention	C0415: Import outside top level
	Error	E0401: Import error
Fatal Errors	Fatal	All fatal error messages

3.4 Data Analysis

We analyzed the collected data using both quantitative and qualitative methods to provide a comprehensive answer to the research questions. This section details the metrics, visualization techniques, statistical tests, and analytical approaches employed to interpret the experimental results.

3.4.1 Quantitative Metrics

To statistically measure and compare performance across models, prompts, and the canonical solutions, we used the following metrics. For each metric, we calculate both individual values (per model-prompt-benchmark combination) and aggregate statistics (overall, benchmark-specific, and model-specific means) to enable multi-level comparisons.

Syntactic Correctness Rate The percentage of generated solutions that successfully parse into an AST without syntax errors. This metric measures the basic well-formedness of generated code across different models and prompting techniques, calculated for each model-prompt-benchmark combination. Solutions that fail to parse are excluded from subsequent smell analysis, as static analysis tools cannot operate on syntactically invalid code.

Smell Density The average number of Pylint and Bandit messages generated per syntactically correct solution. This metric quantifies the overall internal quality of generated code. We calculate smell density separately for each smell category (*Convention, Refactor, Warning, Error, Security*) and in aggregate.

Smelly Samples Percentage The percentage of syntactically valid samples that contained at least one code smell. This metric helps determine how widespread code smells are across the generated code base, distinguishing between pervasive (high percentage) and isolated problematic samples (low percentage but high smell density in affected samples).

3.4.2 Distribution Analysis

To analyze the distribution and composition of code smells, we employed both ranking and visualization to reveal global patterns and model-specific characteristics.

Frequency Rankings We identified the most frequently occurring code smells across different levels of aggregation (models, prompts, and benchmarks) to highlight the most common issues overall.

Distribution Visualization To visualize the distribution characteristics, we employed box plots to reveal median values, Interquartile Range (IQR), and outlier patterns. Box plots

enable us to distinguish between systematic quality degradation (high median, wide IQR) and unreliable generation (normal median, extreme outliers).

3.4.3 Statistical Testing

To provide statistical validation for our findings, we employed a structured non-parametric testing framework. We selected non-parametric tests because the code smell data is discrete, exhibits non-normal distributions, and contains outliers—violating the assumptions of parametric tests such as Analysis of Variance (ANOVA).

Analysis of Model Performance (RQ1.1) To determine if model architecture significantly influences code quality, we conducted the following analyses:

1. **Kruskal-Wallis H test**: We performed this test separately for each benchmark (CoderEval and BigCodeBench-Hard) to assess whether the three models produced significantly different smell densities under Zero-Shot prompting.
2. **Post-hoc pairwise comparisons**: Where the Kruskal-Wallis H test revealed significant differences ($p < 0.05$), we performed pairwise Mann-Whitney U tests with Bonferroni correction ($\alpha = 0.0167$, i.e., $0.05/3$ for the three pairwise comparisons) to identify which specific model pairs differed significantly.
3. **Impact of task complexity**: To assess whether smell prevalence varies with benchmark difficulty, we compared the smell distributions of each model between CoderEval and BigCodeBench-Hard using the Mann-Whitney U test. This analysis reveals whether the quality advantage of specific models is task-dependent or universal.

Comparison Against Canonical Solutions (RQ1.2) Comparing LLM-generated code with human-written canonical solutions posed methodological challenges, particularly in CoderEval, where many canonical solutions depend on broader project contexts. We addressed this through a multi-stage analytical approach:

1. **Intersection filter (CoderEval)**: To ensure fair comparison, we restricted the CoderEval analysis to the 146 tasks where the human solution was syntactically valid and analyzable by our tools. For BigCodeBench-Hard, which consists of self-contained tasks, all canonical solutions were analyzable without filtering.
2. **Kruskal-Wallis H test**: We conducted this test across four groups (Canonical, Phi-3-mini, Phi-4, Qwen2.5-Coder) to determine if the source of code (human vs. LLMs) influenced overall smell density.
3. **Post-hoc Mann-Whitney U tests**: Upon observing highly significant differences in the Kruskal-Wallis H test, we performed pairwise comparisons between the canonical solutions and each model individually, applying a strict Bonferroni correction ($\alpha = 0.0167$ for three comparisons) to ensure statistical robustness against Type I error inflation.

4. **Quality profile analysis**: To investigate whether human developers and LLMs exhibit fundamentally different "smell profiles," we analyzed the distribution of five smell categories. We employed the Chi-Square Test of Independence (χ^2) to determine whether the distribution of these smell types is statistically dependent on the code's origin. This test reveals whether humans and LLMs make qualitatively different types of mistakes.

5. **Effect size measurement**: Given the large sample sizes (hundreds of code samples per group), statistical significance alone can be misleading, as even trivial differences may achieve $p < 0.05$. Therefore, we calculated Cramér's V to measure the practical magnitude of distributional differences, where $V < 0.3$ indicates a small effect, $0.3 \leq V < 0.5$ indicates a medium effect, and $V \geq 0.5$ indicates a large effect following Cohen's conventions [16]. Only large effect sizes ($V \geq 0.5$) indicate strong divergence in the distribution structure that is practically meaningful.

Security Pattern Analysis (RQ1.2) To evaluate whether LLMs exhibit distinct security vulnerabilities compared to human developers, we analyzed security-smell patterns using a matched-samples design. We identified 278 tasks (from the combined 378 benchmark tasks) where all four code sources—Phi-3-mini, Phi-4, Qwen2.5-Coder, and the canonical human solution—successfully generated analyzable code. This intersection filtering ensures fair comparison by eliminating tasks where generation failures or syntactic invalidity would confound the security analysis.

For each of these 278 tasks, we recorded the binary presence/absence of specific Bandit warnings across all four sources, yielding a matched 4×278 response matrix. We then applied:

1. **Cochran's Q test**: A non-parametric test for related samples to assess whether security smell prevalence differs significantly across code sources for each specific Bandit warning type. This test determines whether the probability of generating a particular security smell varies by code source.

2. **Post-hoc McNemar tests**: For security smells showing significant heterogeneity in Cochran's Q test ($p < 0.05$), we performed pairwise McNemar tests comparing each model against the canonical solutions. The McNemar test is appropriate for paired binary data and assesses whether one code source produces more instances of a given security smell than another. We applied Bonferroni correction ($\alpha = 0.0167$ for three comparisons per smell type) to maintain family-wise error control.

Evaluation of Prompt Engineering Efficacy (RQ2) To assess whether different prompting strategies significantly affect code quality, we conducted:

1. **Kruskal-Wallis H test**: We performed this test for each model-benchmark combination to determine if smell density varied significantly across the five prompting techniques.

2. **Post-hoc pairwise comparisons**: For cases where the Kruskal-Wallis H test indicated significant differences ($p < 0.05$), we performed Mann-Whitney U tests comparing each of the four intervention prompts (Quality-Focused, Persona-Based, CoT, RCI)

against the Zero-Shot baseline. We applied Bonferroni correction ($\alpha = 0.0125$ for four comparisons) to control for multiple testing. This approach evaluates whether each prompting strategy produces a statistically significant improvement or degradation relative to Zero-Shot.

3. **Qualitative smell profile shifts**: Beyond overall counts, we analyzed how specific smell categories shifted under different prompting strategies to understand the nature of quality improvements or degradations (RQ2.2).

3.4.4 Exploratory Qualitative Analysis

While quantitative metrics provide a high-level overview of smell prevalence and statistical significance, they fail to explain the mechanisms underlying these phenomena. To gain deeper, contextual insights into why specific flaws occur in LLM-generated code, we conducted qualitative case study analysis through manual inspection of source code for specific, high-impact smells and prompting behaviors. This analysis involved three investigative approaches:

- **Failure mode analysis**: To identify recurring failure patterns, we examined different instances of critical code smells across generated solutions, investigating whether these issues result from knowledge deficits of LLMs or from the sequential nature of the generation process itself.
- **Problematic technique analysis**: To understand why certain prompting techniques consistently increased smell density despite their sophisticated mechanisms, we conducted comparative case studies examining the root causes of quality degradation.
- **Successful technique analysis**: To validate the efficacy of prompting strategies that improved code quality, we examined instances where these approaches demonstrably enhanced code structure.

These case studies provide concrete evidence to help understand the models' underlying generation behavior and the specific mechanisms by which different prompting strategies affect code quality. The qualitative findings are presented in detail in Chapter 5, complementing the quantitative results from Chapter 4.

4

Results

This chapter presents the empirical findings of our investigation into the internal quality of code generated by open-source LLMs. First, we establish a quantitative baseline for code smell prevalence in standard generation scenarios and compare LLM outputs with human-written code (RQ1). Second, we evaluate the effectiveness of various prompt engineering techniques in modulating these code smells (RQ2). Finally, we synthesize findings across all experiments.

4.1 Syntactic Validity

Before assessing the internal quality of generated solutions via static analysis, we evaluated their syntactic validity. Our methodology for this validation is detailed in Section 3.3.1.

Our results indicate a high level of syntactic stability across the tested models. As detailed in Table 4.1, we achieved an overall syntactic correctness rate of 98.48%, indicating that the vast majority of generated snippets constitute well-formed Python code. LLMs with larger parameter counts—specifically Phi-4 and Qwen2.5-Coder—demonstrated near-perfect performance, achieving 100% or near-100% validity in multiple prompting configurations.

However, we observed performance variation when employing complex prompting strategies on the smallest model. Phi-3 exhibited the lowest stability, particularly when combined with the RCI technique on BigCodeBench-Hard, where validity dropped to 89.86%. This decline suggests that while RCI encourages self-correction and reasoning, the added complexity may paradoxically increase the likelihood of generating malformed syntax in models with limited context windows. In contrast, Quality-Focused and CoT maintained higher syntactic validity even on Phi-3.

These validity rates provide the foundation for the subsequent code smell analyses in RQ1 and RQ2, where we analyze exclusively the subset of syntactically valid solutions.

4.2 RQ1: Code Smell Prevalence

This section comprises two subsections: (1) RQ1.1 analyzes the frequency and distribution of code smells in standard Zero-Shot generation, providing the baseline from which all other analyses proceed; (2) RQ1.2 contextualizes these findings by comparing LLM-generated code against human-written canonical solutions, revealing how LLM outputs differ in quality characteristics from professional code.

Table 4.1: Syntactic Correctness Rates (Percentage of Parsable Code) Across Three Models and Five Prompting Techniques on Both Evaluation Benchmarks

Model	**Prompt Technique**	**CoderEval (%)**	**BigCodeBench-Hard (%)**
Phi-3-mini	Zero-Shot	93.91	96.62
	Quality-Focused	96.52	97.97
	Persona-Based	97.39	98.65
	CoT	97.83	95.95
	RCI	95.22	89.86
Phi-4	Zero-Shot	99.57	100.00
	Quality-Focused	100.00	99.32
	Persona-Based	100.00	99.32
	CoT	100.00	100.00
	RCI	100.00	100.00
Qwen2.5-Coder	Zero-Shot	99.13	100.00
	Quality-Focused	100.00	100.00
	Persona-Based	100.00	99.32
	CoT	99.13	100.00
	RCI	99.13	99.32

4.2.1 RQ1.1: Smell Frequency and Distribution

To determine the baseline quality of LLM-generated code, we quantified the frequency of code smells across our two benchmarks. Table 4.2 and Table 4.3 present the distribution of smell categories and smell density for each model. Our analysis reveals that code smells are a pervasive issue in LLM-generated code. The majority of generated solutions contain at least one code smell, with the proportion of smelly samples ranging from 52.40% to 78.38%.

Table 4.2: Prevalence of Code Smells in LLM-Generated Code on BigCodeBench-Hard Using Zero-Shot Prompting, Categorized by Message Type (Convention, Refactor, Warning, Error, Security)

Model	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly Samples (%)	Smell Density
Phi-3-mini	51	21	144	39	70	325	74.13	2.27
Phi-4	47	16	121	16	70	270	72.97	1.82
Qwen2.5-Coder	50	19	122	21	66	278	78.38	1.88

Table 4.3: Prevalence of Code Smells in LLM-Generated Code on CoderEval Using Zero-Shot Prompting, Categorized by Message Type (Convention, Refactor, Warning, Error, Security)

Model	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly Samples (%)	Smell Density
Phi-3-mini	50	98	104	87	28	367	62.96	1.70
Phi-4	32	96	88	20	35	271	52.40	1.18
Qwen2.5-Coder	34	37	204	91	30	396	53.95	1.74

Impact of Task Complexity Task complexity plays a crucial role in the manifestation of code smells. As indicated in Table 4.2 and Table 4.3, we observe a marked increase in smell frequency within the BigCodeBench-Hard dataset compared to CoderEval across all models. The pairwise Mann-Whitney U comparisons yielded highly significant results for all three models (all $p < 0.002$). Specifically, Phi-3-mini ($U = 12454.0$, $p \approx 0.001$) and Phi-4 ($U = 12633.0$, $p < 0.001$) showed substantial quality degradation. Crucially, Qwen2.5-Coder also exhibited a statistically significant increase in smell density ($U = 12589.0$, $p < 0.001$). The implication is that as tasks become more cognitively demanding, models allocate cognitive resources to solving the problem at the expense of code maintainability.

Model Architecture Effects Model architecture significantly influences code quality. On the CoderEval benchmark, the Kruskal-Wallis H test indicated a statistically significant

difference in smell prevalence ($H(2) = 6.43$, $p = 0.040$). Post-hoc analysis revealed that Phi-4 produced significantly fewer code smells than Phi-3-mini ($U = 21559.0$, adjusted $p \approx 0.04$). Phi-4 achieved the lowest smell density at 1.18 smells per sample, compared to 1.70 for Phi-3-mini. Interestingly, the significantly larger Qwen2.5-Coder-32B produced a raw average of 1.74 smells per sample, slightly exceeding Phi-3-mini's performance. However, the comparisons between Qwen2.5-Coder and the other models failed to reach statistical significance after correction.

Conversely, on the BigCodeBench-Hard benchmark, no significant differences between any pair of models after controlling for multiple testing ($H(2) = 1.67$, $p = 0.43$). To visually unpack the distribution underlying these non-significant results on BigCodeBench-Hard, we present a box plot analysis in Figure 4.1. The plot corroborates the uniformity of the central tendency across architectures, with all three models sharing a median smell density of 1.0 per task. While Phi-4 maintains the tightest control (Mean = 1.44, $Q_3 = 2.00$), Qwen2.5-Coder exhibits significant instability, characterized by a long tail of extreme outliers exceeding 20 smells per task. Conversely, Phi-3-mini displays a wider interquartile range ($Q_3 = 3.00$), indicating a generalized tendency toward lower quality. This high variance—particularly the extreme outliers in Qwen2.5-Coder—likely prevents the statistical tests from detecting a significant separation between the groups, despite the observable differences in mean performance.

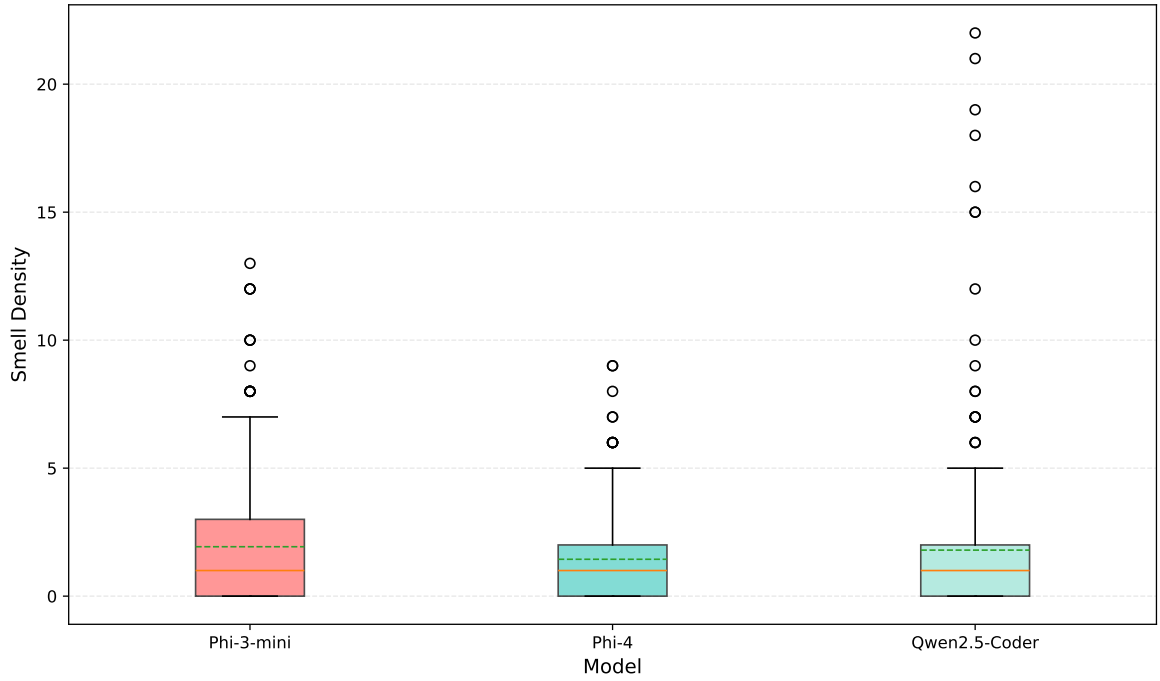

Figure 4.1: Distribution of Code Smell Density Across Three Open-Source Models (Phi-3-mini, Phi-4, Qwen2.5-Coder) on BigCodeBench-Hard Using Zero-Shot Prompting

A granular analysis reveals that each architecture exhibits a distinct "smell profile," which we detail in Table 4.4. The most striking finding is the structural fragility of Qwen2.5-Coder. Despite being the largest model with high syntactic validity, it exhibits a massive anomaly: *W0311: Bad indentation* is its second-most-frequent flaw, a virtually non-existent issue in the

Phi family. In Python, where whitespace determines block structure, this is not merely a cosmetic annoyance but a significant maintenance risk. The persistence of this smell suggests that while Qwen2.5-Coder produces parsable code, it likely mixes tabs and spaces.

Table 4.4: Five Most Frequent Code Smells per Model (Phi-3-mini, Phi-4, Qwen2.5-Coder) in Zero-Shot Generation Showing Smell Type, Message, and Occurrence Count

Model	Rank	Type	Message	Count
Phi-3-mini	1	Convention	C0301: Line too long	90
	2	Error	E0602: Undefined variable	90
	3	Refactor	R0903: Too few public methods	61
	4	Warning	W0612: Unused variable	54
	5	Warning	W0613: Unused argument	35
Phi-4	1	Convention	C0301: Line too long	71
	2	Refactor	R0903: Too few public methods	55
	3	Warning	W0612: Unused variable	35
	4	Warning	W1514: Unspecified encoding	32
	5	Warning	W0613: Unused argument	27
Qwen2.5-Coder	1	Error	E0602: Undefined variable	90
	2	Warning	W0311: Bad indentation	87
	3	Convention	C0301: Line too long	78
	4	Warning	W0612: Unused variable	54
	5	Warning	W0212: Protected access	37

Smell Category Analysis To understand the quality characteristics of LLM-generated code, we analyze the distribution and frequency of detected code smells, examining which smells are most prevalent and what they reveal about model behavior. Table 4.5 lists the ten most frequently occurring code smells across all syntactically valid solutions generated using the Zero-Shot prompt.

The most frequent issue, *C0301: Line too long*, suggests that models prioritize information density over readability; even though this is a lower-severity issue than *E0602: Undefined variable*, it still affects code readability and the maintenance burden.

The second most common issue is *E0602: Undefined variable*. We note that while *E0602: Undefined variable* is flagged as an *Error*, a subset of these instances in the CoderEval benchmark may represent valid references to class attributes or project-level constants that are invisible to the static analyzer in the isolated function scope. Therefore, this metric represents an upper bound on logical errors.

We also observe high counts for *W0612: Unused Variable* and *W0613: Unused Argument*. The prevalence of these smells indicates that models struggle to efficiently identify the variables they declare. Models frequently initialize variables or iterate over data structures

without subsequently using them, creating "dead code" that adds cognitive overhead for developers and obscures the code's intent.

Other refactoring smells, such as *R0903: Too few public methods* and *R1705: No else return*, point to suboptimal design choices. The *R0903: Too few public methods* smell typically indicates that a class is being used as a simple data structure. However, its high frequency in this study is likely an artifact of the benchmark design. Since many tasks require generating a single function in isolation, models may wrap that function in a class, which triggers the Pylint messages. This is a valuable reminder that not all detected smells represent genuine quality problems; context is essential for interpretation.

Table 4.5: Ten Most Frequent Code Smells Aggregated Across All Models and Benchmarks in Zero-Shot Generation with Type, Message, and Total Occurrence Count

Rank	Type	Message	Count
1	Convention	C0301: Line too long	238
2	Error	E0602: Undefined variable	196
3	Warning	W0612: Unused variable	143
4	Refactor	R0903: Too few public methods	117
5	Warning	W0311: Bad indentation	97
6	Warning	W0613: Unused argument	94
7	Warning	W1514: Unspecified encoding	90
8	Refactor	R1705: No else return	71
9	Security	B311: Import of random library	70
10	Warning	W0212: Protected access	67

Security Smell Patterns We identified a focused set of security smells detailed in Table 4.6. The most notable finding is *B113: Request Without Timeout*. This smell flags HTTP requests that lack a specified timeout parameter—a practice that can cause applications to hang indefinitely. This suggests that LLMs frequently default to the simplest API implementations (e.g., `requests.get()`) rather than applying defensive coding practices required for production environments. This is a genuine quality concern warranting attention.

In contrast, qualitative inspection suggests that the most frequent smell-*B311: Import of random library*-is primarily driven by the benchmark tasks requirement. This smell indicates that using this module is not suitable for security or cryptographic purposes, as its pseudo-random number generator is predictable. However, many benchmark tasks require random data generation, making the `random` module an appropriate choice for the problem. This smell is a false positive in context.

Similarly, other security issues, such as *B404: Import of subprocess library*, are triggered because Bandit strictly warns about processes spawned to alert developers to potential shell injection risks. Since some benchmark tasks explicitly require interaction with external processes, using `subprocess` is unavoidable. This, too, is more a limitation of Bandit's detection approach than a genuine quality problem in the generated code.

Consequently, we classify *B311: Import of random library* and *B404: Import of subprocess library* as task-induced artifacts rather than genuine vulnerabilities. The high prevalence of these specific codes reflects the benchmark requirements rather than a model's propensity for insecure coding.

Table 4.6: Five Most Frequent Security Smells Detected by Bandit Aggregated Across All Models with Severity Level, and Occurrence Count

Rank	Message	Severity	Count
1	B311: Import of random library	Low	70
2	B113: Request Without Timeout	Medium	47
3	B404: Import of subprocess library	Low	46
4	B603: Subprocess Without Shell Equals True	Low	35
5	B310: Urllib_urlopen	Medium	18

4.2.2 RQ1.2: Comparison with Canonical Solutions

To provide context for the preceding findings and to ground our assessment of LLM-generated code quality in realistic terms, this section addresses RQ1.2: *How does the prevalence of smells in LLM-generated code compare to that in human-written canonical solutions?* This comparison is essential for determining whether the observed code smells are unique artifacts of generative models or are reflective of problems also present in human-written code.

Smell Density Differential The comparison between LLM-generated and human-written code reveals a counterintuitive pattern: quantitatively, LLM code exhibits superior metrics to expert-written canonical solutions. As shown in Table 4.7, human-written canonical solutions contain substantially more code smells across both benchmarks, with an average smell density of 4.27 smells per sample compared to 1.77 for LLMs—representing a 2.4× difference. Moreover, 89.80% of human-written samples contain at least one smell, compared to only 65.8% for LLM-generated code. This finding appears paradoxical: if LLMs are trained on human code repositories, how can they generate code that is systematically cleaner than that of expert developers? The resolution lies in the quality type. The data reveals that LLMs and humans prioritize fundamentally different quality dimensions.

Smell Profile Comparison To understand why LLMs generate fewer total smells, we examine the distribution of smell types. Figure 4.2 presents the proportional distribution of different types of smells. We restricted this visual analysis to the BigCodeBench-Hard dataset to ensure a more precise comparison. As identified in the statistical analysis, the CoderEval benchmark introduces a disproportionate number of false-positive *Error* flags in the canonical solutions due to missing project context. BigCodeBench-Hard, consisting of self-contained tasks, eliminates this artifact.

Table 4.7: Prevalence of Code Smells in Human-Written Canonical Solutions from Both Benchmarks, Categorized by Message Type with Smell Density Metrics

Benchmark	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly Samples (%)	Smell Density
BigCodeBench-Hard	397	22	140	3	62	624	94.59	4.22
CoderEval	118	52	41	399	22	632	84.93	4.33
Combined	**515**	**74**	**181**	**402**	**84**	**1256**	**89.80**	**4.27**

The visualization reveals a distinct profile in quality priorities. For all three LLMs, the distribution is dominated by the *Warning* category. This indicates that while models generally succeed in generating syntactically valid code, they consistently produce structural inefficiencies that leave execution intact but compromise maintainability. In sharp contrast, the human canonical profile is overwhelmingly dominated by the *Convention* category. Crucially, the *Error* category is virtually non-existent in the canonical solutions.

While the quantitative data favors LLMs, the qualitative nature of the smells differs fundamentally. The high smell density in human solutions is driven predominantly by *Convention* violations (e.g., line lengths) and stylistic choices. In contrast, LLM solutions, despite having lower total density, are weighted toward *Error* and *Warning* categories. Thus, while LLMs appear 'cleaner' by metric count, human code remains semantically more robust. The LLM optimizes for local syntax, while the human developer optimizes for global logic.

We also compared the most common smells that appeared in the canonical solutions as detailed in Table 4.8. Both human and LLM authors struggle with *C0301: Line too long* and *E0602: Undefined variable*, which are the top two smells for both groups, though these issues are far more pronounced in the human-written code. The LLM-generated code is characterized by smells like *W0612: Unused Variable*, *R0903: Too few public methods*, and *W0311: Bad indentation*. This profile suggests a tendency to generate structurally naive code—code that runs but carries inefficiencies that complicate understanding and maintenance. In contrast, the smells found in human-written code include *C0209: Consider using f-string*, *W1514: Unspecified encoding* , and *W0102: Dangerous default value*. This profile points to issues of code modernization and subtle, experience-based errors, reflecting code that may be older or written without adherence to the latest best practices.

To confirm that the observed differences in smell profile were statistically significant, we performed pairwise Chi-Square Tests of Independence as specified in Section 3.4.3. The results, summarized in Table 4.9, reveal that the distribution of issue types in canonical solutions differs significantly from every model on both benchmarks (all $p < 0.001$). The effect size analysis (Cramér's V) highlights distinct patterns of divergence, ranging from medium ($V = 0.442$) to large effects ($V = 0.609$).

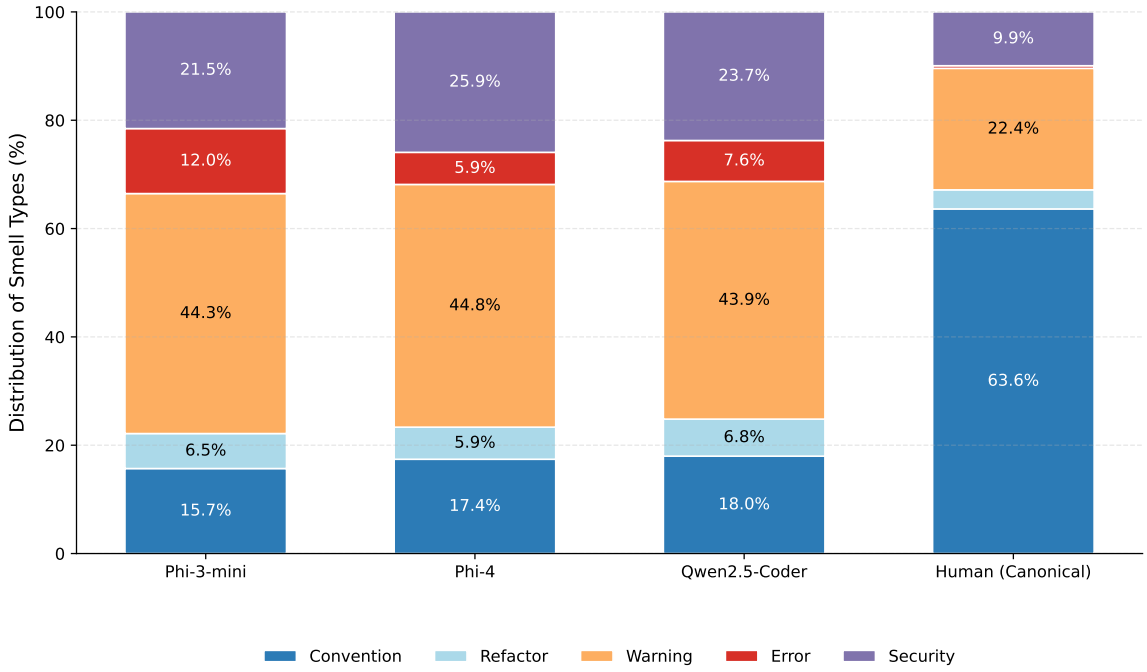

Figure 4.2: Distribution of Code Smell Categories (Convention, Refactor, Warning, Error, Security) Across LLM-Generated Code and Canonical Solutions on BigCodeBench-Hard

Table 4.8: Ten Most Frequent Code Smells Detected in Human-Written Canonical Solutions Across Both Benchmarks with Message Type and Occurrence Count

Rank	Type	Message	Count
1	Convention	C0301: Line too long	460
2	Error	E0602: Undefined variable	397
3	Warning	W0612: Unused variable	52
4	Convention	C0209: Consider using f-string	39
5	Warning	W1514: Unspecified encoding	33
6	Warning	W0707: Raise missing `from`	24
7	Security	B311: Import of random library	24
8	Refactor	R1705: No else return	18
9	Warning	W0102: Dangerous default value	14
10	Security	B404: Import of subprocess library	12

Table 4.9: Chi-Square and Cramér's V Effect Sizes Comparing Code Smell Distribution Profiles Between Human-Written and LLM-Generated Code on Both Benchmarks

Dataset	Comparison	χ^2 (df=4)	p-value	Cramér's V
BigCodeBench-Hard	Human vs. Phi-3-mini	226.97	$< 10^{-47}$	0.489
	Human vs. Phi-4	174.85	$< 10^{-36}$	0.442
	Human vs. Qwen2.5-Coder	177.91	$< 10^{-36}$	0.444
CoderEval	Human vs. Phi-3-mini	214.84	$< 10^{-44}$	0.464
	Human vs. Phi-4	334.42	$< 10^{-70}$	0.609
	Human vs. Qwen2.5-Coder	314.63	$< 10^{-66}$	0.553

Security Smell Comparison To evaluate whether LLMs exhibit distinct security vulnerabilities compared to human developers, we analyzed security-smell occurrence patterns across both datasets, as detailed in Table 4.10.

The analysis revealed no significant differences across code sources for three of the five security smells: *B404: Import of subprocess library*, *B603: Subprocess without shell=True*, and *B310: Urllib_urlopen* all showed nearly identical prevalence rates (all $p > 0.3$). This near-perfect consistency provides strong evidence that these smells reflect benchmark design constraints rather than differences in quality.

For *B311: Import of random library*, we observed light variation across groups ($Q(3) = 10.20$, $p = 0.017$). However, the practical significance of this difference is minimal given the benchmark context where random number generation is explicitly required.

Most critically, *B113: Request Without Timeout* showed the strongest heterogeneity in the entire security analysis ($Q(3) = 24.43$, $p < 0.001$). All three LLMs exhibited substantially elevated prevalence (5.4–5.8%) compared to the canonical baseline (2.5%)—representing a 2.2× increase. Post-hoc McNemar tests confirmed significant differences for all three models on this issue:

- Phi-3-mini vs. Canonical: $\chi^2 = 9.00$, adjusted $p = 0.008$
- Phi-4 vs. Canonical: $\chi^2 = 9.00$, adjusted $p = 0.008$
- Qwen2.5-Coder vs. Canonical: $\chi^2 = 8.00$, adjusted $p = 0.014$

The consistency of this elevation across all models indicates a behavioral pattern. LLMs appear to default to the simplest API usage patterns (e.g., `requests.get(url)`), without incorporating defensive programming practices such as timeout specification that experienced developers apply reflexively.

4.3 RQ2: Prompt Engineering Effects

This section answers our second primary research question: *Can strategic prompt engineering prevent code smells in LLM-generated code?* Having established in RQ1 that code smells are

Table 4.10: Cochran's Q Test Results for Security Smell Prevalence Comparing Human-Written Canonical Solutions Against Three LLM Models Across 278 Matched Tasks

Smell	Phi-3-mini	Phi-4	Qwen2.5-Coder	Canonical	Q	p
B311: Import of random library	4.7	5.8	5.8	6.5	**10.20**	**0.017**
B404: Import of subprocess library	5.0	5.0	5.0	4.3	3.00	0.392
B603: Subprocess shell=True	3.6	3.6	3.2	3.6	0.36	0.948
B310: Urllib_urlopen	2.2	2.2	2.2	2.2	0.00	1.000
B113: Request without timeout	5.8	5.8	5.4	2.5	**24.43**	**0.000**

prevalent in LLM-generated code and that they differ qualitatively from human-written code, we now present our findings on whether developers can actively prevent these code smells through deliberate prompt design.

4.3.1 RQ2.1: Quantitative Impact

To assess the overall effectiveness of prompt engineering in reducing code smells, we generated solutions for all benchmark tasks using each of the five prompting strategies. We present the results visually in Figure 4.3 and Figure 4.4, which display the mean smell density for each model-technique combination across both benchmarks.

On BigCodeBench-Hard, the data shows relatively uniform performance across techniques with high Zero-Shot smell densities. However, CoderEval exhibits greater variation between techniques, with some approaches showing meaningful reductions for Qwen2.5-Coder.

More importantly, we observed three key findings from the results: (1) Quality-Focused shows modest improvements; (2) RCI consistently degrades performance; and (3) effects are far more pronounced on CoderEval than BigCodeBench-Hard. The remainder of this section examines each prompt technique in detail.

Quality-Focused The Quality-Focused prompt, which directly instructs the model to generate clean code and avoid code smells, achieved the most substantial reduction in code smell density across all prompting techniques. On the CoderEval benchmark, this technique reduced the average smell count for Qwen2.5-Coder by 42.5%, from 1.74 smells per sample in the Zero-Shot baseline to 1.00.

However, this benefit is not universal. On BigCodeBench-Hard, the Quality-Focused prompt showed minimal improvement, with smell density remaining at 1.97 smells per sample compared to 1.97 for Zero-Shot. This suggests that on highly complex tasks, explicit quality instructions provide limited benefit. The model's capacity is already exhausted by the task's complexity, leaving minimal cognitive resources for incorporating quality guidance.

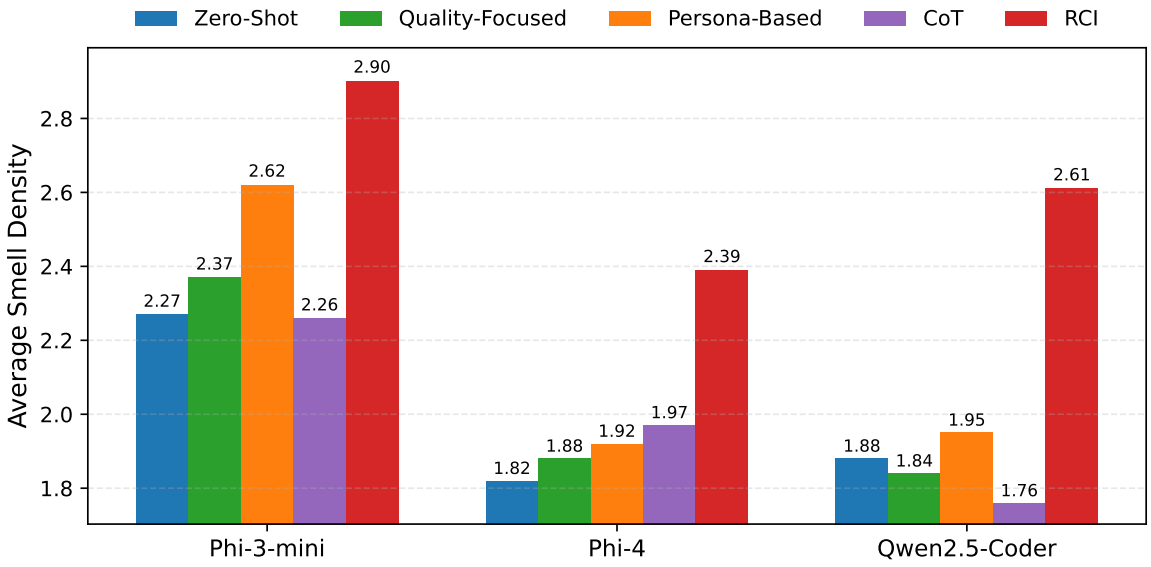

Figure 4.3: Impact of Five Prompting Techniques (Zero-Shot, Quality-Focused, Persona-Based, CoT, RCI) on Code Smell Density Across Three Models on BigCodeBench-Hard

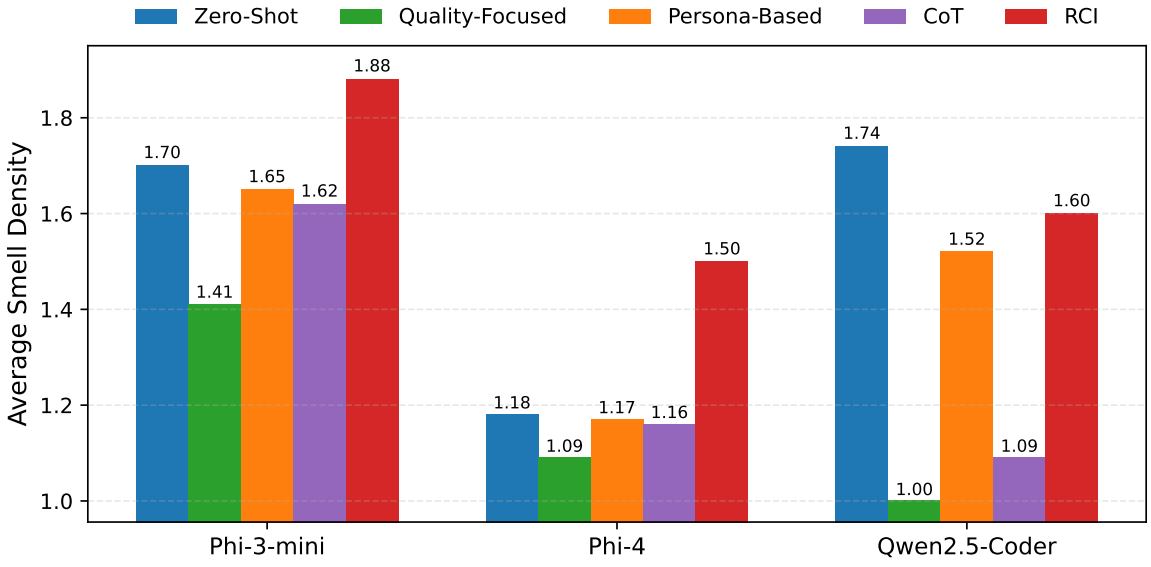

Figure 4.4: Impact of Five Prompting Techniques (Zero-Shot, Quality-Focused, Persona-Based, CoT, RCI) on Code Smell Density Across Three Models on CoderEval

CoT The CoT prompt demonstrated similarly promising results as Quality-Focused. On CoderEval, CoT reduced smell density by 37.4% (from 1.74 to 1.09 smells per sample) for Qwen2.5-Code. This suggests that encouraging the model to articulate its reasoning process before generating code can lead to more deliberate design choices that align with quality standards.

However, this effect was not consistent across all experimental conditions. On the more complex BigCodeBench-Hard benchmark, CoT showed minimal to no improvement and, in some cases, led to slight increases in smell density. For instance, with Phi-4 on BigCodeBench-Hard, CoT increased smell density from 1.93 to 2.07 smells per sample. This degradation mirrors the Quality-Focused pattern: on complex tasks where models already struggle, adding reasoning steps may introduce additional opportunities for errors rather than improvements.

Persona-Based The Persona-Based prompt, which instructs the model to adopt the role of a software quality expert, proved largely ineffective across all models and benchmarks. This technique failed to produce meaningful reductions in code smells and, in several cases, resulted in marginal increases. For example, with Phi-3-mini on BigCodeBench-Hard, the Persona-Based prompt increased smell density from 2.34 to 2.72 smells per sample. This finding challenges the intuitive assumption that role-playing prompts can meaningfully alter the fundamental patterns of code generation, thereby improving measurable quality metrics.

The ineffectiveness of this approach aligns with emerging research suggesting that persona prompts may have limited and inconsistent effects on objective technical tasks [26, 89].

RCI The most striking and counterintuitive finding of this analysis concerns the RCI technique. This multi-step prompting strategy consistently increased smell density across all models and benchmarks.

With Qwen2.5-Coder on BigCodeBench-Hard, RCI increased the average smell density by 38.7% over Zero-Shot, from 1.97 to 2.71 smells per sample. On CoderEval, the trend persisted, with Phi-4 increasing from 1.18 to 1.52 smells per sample. Most critically, in nearly every tested scenario, RCI produced the highest smell density of any prompting technique, indicating that this approach is systematically counterproductive.

This finding demonstrates that prompting sophistication does not automatically lead to better outcomes—sometimes, simpler approaches are more effective precisely because they introduce fewer opportunities for error.

Statistical Validation of Prompting Effects To determine whether the observed differences in smell density across prompting techniques were statistically significant, we conducted Kruskal-Wallis H tests for each model-benchmark combination, as specified in Chapter 3.

The results, presented in Table 4.11, reveal that the prompting technique significantly influences code quality only under specific conditions. For Qwen2.5-Coder, we observed highly significant differences on both CoderEval ($H(4) = 16.17$, $p = 0.0028$) and BigCodeBench-Hard ($H(4) = 20.33$, $p = 0.0004$). In contrast, Phi-4 exhibited no statistically significant differences on either benchmark. Phi-3-mini showed a significant effect on CoderEval

($H(4) = 9.93$, $p = 0.0416$) but not on BigCodeBench-Hard ($H(4) = 6.56$, $p = 0.1614$). This suggests that the impact of prompt engineering is strongly dependent on both the model architecture and task complexity.

Table 4.11: Kruskal-Wallis H Test Results Evaluating Statistical Significance of Prompting Technique Effects on Code Smell Density for Each Model-Benchmark Combination

Model	**Benchmark**	**H-statistic**	**p-value**	**Significant ($p < 0.05$)**
Phi-3-mini	CoderEval	9.93	0.0416	True
Phi-3-mini	BigCodeBench-Hard	6.56	0.1614	False
Phi-4	CoderEval	9.10	0.0587	False
Phi-4	BigCodeBench-Hard	8.33	0.0801	False
Qwen2.5-Coder	CoderEval	16.17	0.0028	True
Qwen2.5-Coder	BigCodeBench-Hard	20.33	0.0004	True

Post-Hoc Pairwise Comparisons To identify which specific prompting techniques differ significantly, we performed post-hoc pairwise Mann-Whitney U tests with Bonferroni correction. Table 4.12 presents the results for model-benchmark combinations in which the Kruskal-Wallis H test revealed significant effects.

For Qwen2.5-Coder-32B on CoderEval, Quality-Focused showed significant improvements over Zero-Shot ($U = 22170.0$, adjusted $p = 0.012$). In contrast, RCI showed no significant difference despite its higher raw smell count. CoT and Persona-Based similarly failed to reach significance.

On BigCodeBench-Hard, the pattern shifts dramatically. Quality-Focused did not reach the corrected threshold. Most critically, RCI showed a significant increase in smell density ($U = 13275.5$, adjusted $p = 0.002$), providing statistical confirmation of its detrimental effect on complex tasks.

Distribution Analysis To visualize these distributional differences, Figure 4.5 and Figure 4.6 present box plots comparing smell density across all five prompting techniques for Qwen2.5-Coder on both benchmarks.

The visualizations reveal key distributional patterns. On CoderEval, the median values for Zero-Shot, Quality-Focused, and CoT remain similar (ranging from 1.0 to 2.0 smells per sample). In contrast, RCI exhibits both a higher median and substantially greater variance, with outliers exceeding eight smells per sample. This pattern indicates that RCI's degradation is not driven by a few extreme cases but rather reflects a systematic shift across the entire distribution toward higher smell density. On BigCodeBench-Hard, the spread is more uniform across Zero-Shot, Quality-Focused, CoT, and Persona-Based techniques, but RCI maintains consistently elevated values throughout its distribution, confirming that its degrading effect is systematic rather than sporadic.

Table 4.12: Post-Hoc Mann-Whitney U Pairwise Comparisons of Four Prompting Techniques Against Zero-Shot Baseline with Bonferroni-Corrected Significance Values

Model	**Benchmark**	**Prompt**	**U-statistic**	**Adjusted p-value**	**Effect Direction**
Phi-3-mini	CoderEval	Quality-Focused	21036.5	1.000	No sig. diff.
Phi-3-mini	CoderEval	Persona-Based	21189.5	1.000	No sig. diff.
Phi-3-mini	CoderEval	CoT	23191.5	1.000	No sig. diff.
Phi-3-mini	CoderEval	RCI	24112.0	0.348	No sig. diff.
Qwen2.5-Coder	CoderEval	Quality-Focused	22170.0	0.012	Improvement
Qwen2.5-Coder	CoderEval	Persona-Based	25190.0	1.000	No sig. diff.
Qwen2.5-Coder	CoderEval	CoT	23359.5	0.256	No sig. diff.
Qwen2.5-Coder	CoderEval	RCI	26586.5	1.000	No sig. diff.
Qwen2.5-Coder	BigCodeBench-Hard	Quality-Focused	10623.5	1.000	No sig. diff.
Qwen2.5-Coder	BigCodeBench-Hard	Persona-Based	11106.5	1.000	No sig. diff.
Qwen2.5-Coder	BigCodeBench-Hard	CoT	10499.0	1.000	No sig. diff.
Qwen2.5-Coder	BigCodeBench-Hard	RCI	13275.5	0.002	Degradation

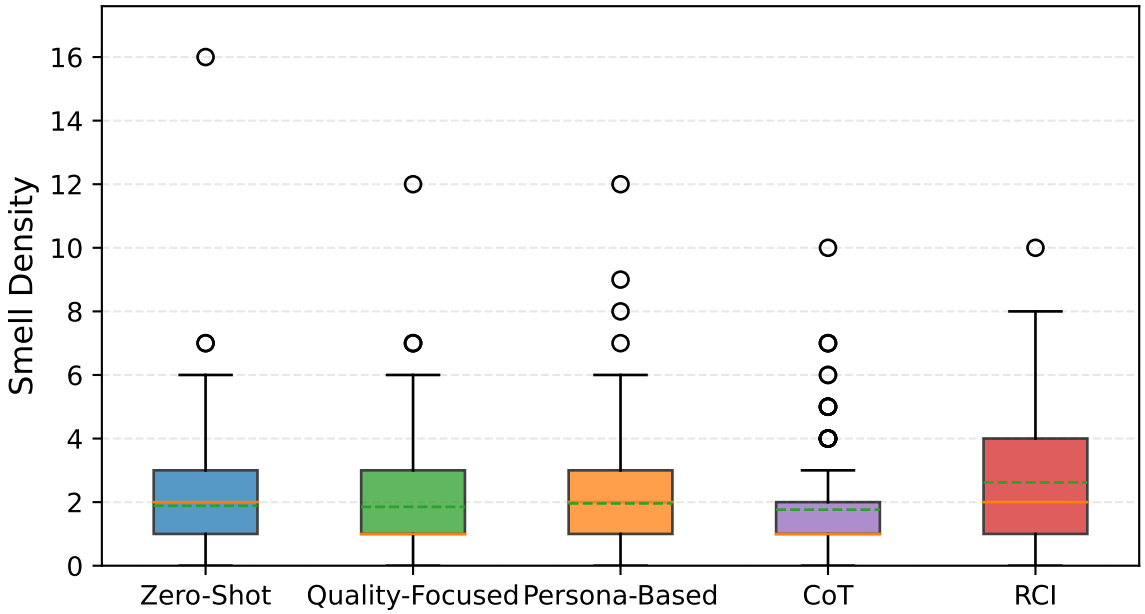

Figure 4.5: Distribution of Code Smell Density Across Five Prompting Techniques for Qwen2.5-Coder on BigCodeBench-Hard Showing Median, Interquartile Range, and Outliers

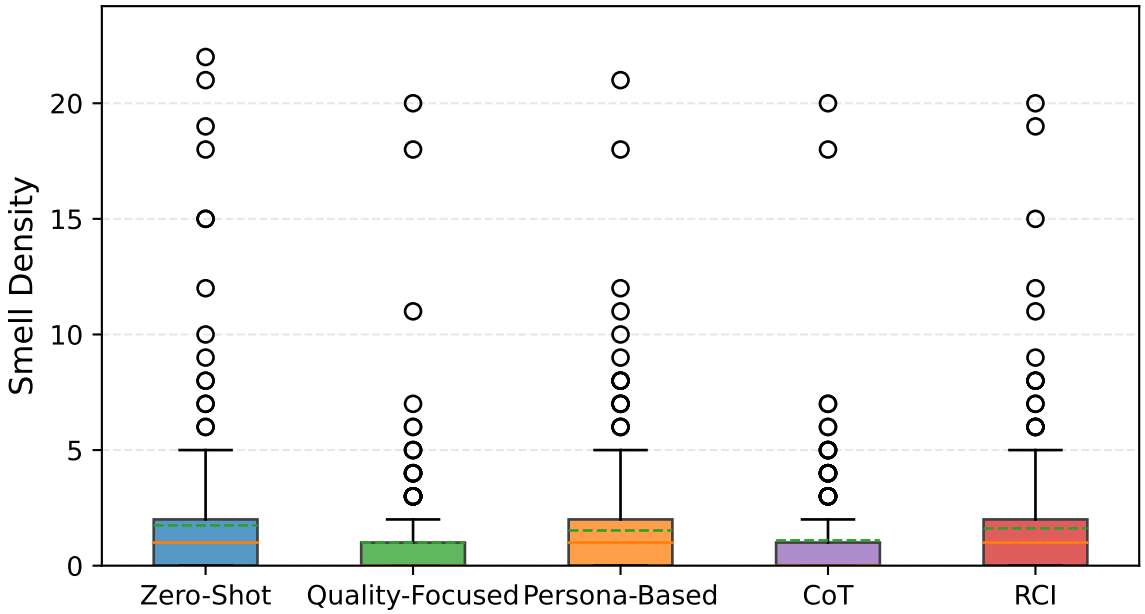

Figure 4.6: Distribution of Code Smell Density Across Five Prompting Techniques for Qwen2.5-Coder on CoderEval Showing Median, Interquartile Range, and Outliers

4.3.2 RQ2.2: Qualitative Profile Shifts

While RQ2.1 established the magnitude of smell reduction—or lack thereof—it is equally critical to understand the morphology of these changes. We analyzed how specific smells shifted under Quality-Focused and RCI to understand the nature of quality improvements or degradations.

Quality-Focused vs. Zero-Shot As established in RQ1, the Zero-Shot baseline for Qwen2.5-Coder exhibited a distinct formatting bias: *W0311: Bad indentation* was its second-most frequent issue (87 occurrences), alongside a high volume of warnings. Applying the Quality-Focused prompt led to a dramatic reduction in these smells.

As shown in Table 4.13, the *W0311: Bad indentation* smell was effectively eradicated (87 → 0). However, the *E0602: Undefined variable* smell remained the Rank 1 issue, dropping from 89 to 65 occurrences. This finding suggests that Prompt engineering is highly effective at enforcing code hygiene (removing indentation errors and unused arguments) and formatting. However, it cannot easily correct hallucinations rooted in the model's training distribution.

Table 4.13: Comparison of Three Most Frequent Code Smells for Qwen2.5-Coder on CoderEval Between Zero-Shot and Quality-Focused Prompting Strategies

Strategy	Rank	Type	Message	Count
Zero-Shot	1	Error	E0602: Undefined variable	89
	2	Warning	W0311: Bad indentation	87
	3	Warning	W0212: Protected access	37
Quality-Focused	1	Error	E0602: Undefined variable	65
	2	Warning	W0212: Protected access	30
	3	Warning	W0613: Unused argument	30

RCI vs Zero-Shot Our analysis of the RCI technique offers a precise explanation for its quantitative failure. As detailed in Table 4.14 and Table 4.15, the *C0301: Line too long* smell exploded in frequency on BigCodeBench-Hard for every tested model. For Phi-3-mini, instances rose by 36.7% (49 → 67), and for Qwen2.5, by 73.5% (49 → 85).

Furthermore, RCI introduced *Refactor* and *Warning* smells that were absent in the baseline rankings. Notably, *W0707: Raise missing from* appeared as the Rank 2 issue for Qwen2.5-Coder (rising from 7 to 48 occurrences), while Phi-3-mini suffered from *W0621: Redefined outer name* issues (rising from 18 to 46). The morphological evidence confirms what quantitative metrics suggested: RCI degrades quality by introducing verbosity and hallucinated refactoring complexity (e.g., improper exception chaining and variable shadowing).

Table 4.14: Comparison of Three Most Frequent Code Smells Between Zero-Shot and RCI Prompting for Phi-3-mini on BigCodeBench-Hard

Strategy	Rank	Type	Message	Count
Zero-Shot	1	Convention	C0301: Line too long	49
	2	Warning	W0612: Unused variable	37
	3	Error	E0602: Undefined variable	23
RCI	1	Convention	C0301: Line too long	67
	2	Warning	W0621: Redefined outer name	46
	3	Warning	W0612: Unused Variable	41

Table 4.15: Comparison of Three Most Frequent Code Smells Between Zero-Shot and RCI Prompting for Qwen2.5-Coder on BigCodeBench-Hard

Strategy	Rank	Type	Message	Count
Zero-Shot	1	Convention	C0301: Line too long	49
	2	Warning	W0612: Unused variable	49
	3	Warning	W1514: Unspecified encoding	22
RCI	1	Convention	C0301: Line too long	85
	2	Warning	W0707: Raise missing from	48
	3	Warning	W0612: Unused Variable	40

4.4 Summary of Findings

4.4.1 RQ1: Code Smell Prevalence

RQ1.1: Frequency and Distribution Our analysis revealed that code smells are pervasive in LLM-generated code, with 52% - 78% of syntactically valid solutions containing at least one code smell. Smell density ranged from 1.18 to 2.27 smells per solution, depending on the model and benchmark. Phi-4 consistently achieved the lowest smell density, outperforming both Phi-3-mini and Qwen2.5-Coder.

The most prevalent issues included *C0301: Line too long* (*Convention*), *E0602: Undefined variable* (Error) and *W0612: Unused Variable* (Warning). Critically, qualitative analysis revealed that *E0602: Undefined variable* likely stems from probabilistic typos, API namespace confusion, case-sensitivity drift, and variable shadowing, rather than a lack of programming knowledge.

We identified genuine security risks, most notably the systematic failure to define API timeouts (*B113: Request Without Timeout*), while other detected security flags were essentially artifacts of task requirements.

RQ1.2: Comparison with canonical solutions Contrary to expectations, LLM-generated code contained quantitatively fewer smells than human-written canonical solutions (1.77 vs. 4.27 average smell density). However, chi-square tests revealed fundamentally different distributions of smells. Human code exhibited predominantly *Convention* violations, while LLM code suffered from *Warning* and *Error* categories.

4.4.2 RQ2: Prompt Engineering Efficacy

RQ2.1: Quantitative Impact Prompt engineering effects varied significantly by technique. The Quality-Focused prompt achieved modest but consistent smell reductions (7%- 15% improvement over Zero-Shot). Persona-Based prompts ("Act as a software quality expert") showed minimal impact. CoT prompting produced mixed results, sometimes increasing smell density due to added complexity.

Most critically, RCI—the most sophisticated technique tested—consistently degraded quality across all models and benchmarks. Statistical analysis confirmed that RCI significantly increased smell density compared to Zero-Shot baselines. This contradicts the assumption that self-correction improves output quality.

RQ2.2: Qualitative Shifts in Smell Distribution Different prompting techniques altered not only the quantity of smell but also its types. Quality-Focused prompts successfully reduced mechanical issues (*W0612: Unused Variable*, *W0311: Bad indentation*) by activating latent knowledge of coding standards. However, RCI introduced a "verbosity penalty"—generating verbose code with excessive try-except blocks and hallucinated abstractions (undefined constants, non-existent helper functions).

5 Discussion

This chapter extends the usual scope of a discussion section—which in many theses is primarily focused on interpreting quantitative results—by including additional analyses aimed at better understanding why those results arise. We begin by interpreting the empirical findings presented in Chapter 4. While the quantitative analysis established high code smell prevalence (RQ1) and inconsistent, sometimes counterproductive, prompting effects (RQ2), these metrics alone cannot fully reveal the underlying mechanisms. To help address this limitation, we qualitatively examine specific failure patterns, illustrating why models with near-perfect parsing rates can still produce low-quality code. We then analyze how different prompting strategies mechanistically shape the generated code. Finally, we address the threats to validity that limit the generalizability of our findings.

5.1 RQ1: Syntax-Logic Gap

As shown in Chapter 4, syntactically valid solutions still exhibited a high prevalence of code smells, with *E0602: Undefined variable* emerging as one of the most frequent and impactful issues across models and benchmarks. This apparent contradiction—valid syntax paired with substantial semantic and structural problems—has also been noted in recent research on LLMs [54, 72, 83], suggesting that strong surface-level fluency can mask inconsistencies in deeper program logic and reinforcing that the phenomenon observed here is not unique to this study.

A defining characteristic of the LLMs evaluated in this study is their mastery of Python syntax paired with fragility in program structure. The prevalence of *E0602: Undefined variable* errors indicates that while models know how to write code, they often forget what they are writing about. Unlike a compiler that uses a symbol table to verify existence, the LLM predicts the next sub-word token based on probability distributions over its training corpus.

This disconnect manifests in multiple forms: probabilistic token slips that introduce typos, API knowledge without namespace awareness, and variable references that drift in casing or scope. Critically, these failures are not due to a lack of programming knowledge—the models demonstrate sophisticated understanding of algorithms, data structures, and library capabilities. Instead, they stem from the autoregressive generation process itself, in which each token is predicted independently, without guaranteeing consistency with earlier choices. To understand the specific mechanisms underlying this disconnect, we conducted a manual inspection of failure cases.

5.1.1 Probabilistic Typos

One of the most surprising failure modes observed is the introduction of typos in variable names—an error class almost non-existent in modern IDEs with autocomplete, yet prevalent in probabilistic generation.

As illustrated in Listing 5.1, the model correctly defines the constant `PEOPLE_COUNT`. However, in a subsequent list comprehension, it references `PEOPEN_COUNT`. This error highlights the stochastic nature of LLM decoding. The token "PEN" likely shared a high probability mass with "PLE" in this context, leading to a typo that renders the code inexecutable.

Listing 5.1: Example of a Probabilistic Typo: Phi-3-mini Hallucinates `PEOPEN_COUNT` Instead of the Defined `PEOPLE_COUNT`

```
def task_func(filename):
    data = []
    names = [f'Person{i}' for i in range(PEOPLE_COUNT)]
    ages = [random.randint(18, 70) for _ in range(PEOPEN_COUNT)]
    heights = [random.uniform(1.5, 2.0) for _ in range(PEOPLE_COUNT)]
    weights = [random.uniform(50, 100) for _ in range(PEOPLE_COUNT)]

    for name, age, height, weight in zip(names, ages, heights, weights):
        data.append([name, age, height, weight])

    avg_age = round(mean(ages), 2)
    avg_height = round(mean(heights), 2)
    avg_weight = round(mean(weights), 2)

    data.append(['Average', avg_age, avg_height, avg_weight])

    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(COLUMNS)
        writer.writerows(data)

    return os.path.abspath(filename)
```

5.1.2 API Hallucinations

A second recurring pattern, identified through close qualitative inspection of the detected smells, involves the correct retrieval of function names but the failure to respect namespace constraints. This phenomenon, which we term "API Amnesia," accounts for a significant portion of the *E0602: Undefined variable* errors observed in the Zero-Shot baseline.

Listing 5.2 demonstrates a case where the model attempts to use `randint` directly. While the logic is sound, the function resides within the `random` module (which was imported). The model failed to qualify the call as `random.randint` or to explicitly import the function.

Similarly, Listing 5.3 shows a call to `random_seed(seed)`. The model likely conflated the standard `random.seed()` method with a training pattern where a helper function named `random_seed` might have existed. These errors suggest that while LLMs possess strong knowledge of library capabilities, they struggle with the precise syntactic requirements of module namespaces.

Listing 5.2: Failure to Call `random.randint`

```
import pandas as pd
from datetime import datetime, timedelta
import random

def task_func(start_date=datetime(2020, 1, 1), end_date=datetime(2020, 12, 31), seed
    =42):
    if not isinstance(start_date, datetime) or not isinstance(end_date, datetime):
        raise ValueError("start_date and end_date must be datetime instances")
    if start_date > end_date:
        raise ValueError("start_date must be before or equal to end_date")

    random.seed(seed)
    date_range = [start_date + timedelta(days=randint(0, (end_date - start_date).days)
        ) for _ in range((end_date - start_date).days + 1)]
    return pd.Series(date_range)
```

5.1.3 Case Sensitivity Errors

A distinct category of *E0602: Undefined variable* arises when the model captures the semantic meaning of a variable but fails to adhere to its specific syntactic casing.

Listing 5.4 illustrates a clear example of this phenomenon. The model correctly defines a global constant `NUMBERS` in uppercase, following standard Python conventions for constants. It even references this constant correctly in the list comprehension: `random.choice(NUMBERS)`. However, merely two lines later, the model attempts to pass the variable to a plotting function as `numbers` (lowercase).

This error suggests that the model's internal representation of the code focuses on the concept of the variable ("the list of numbers") rather than its rigid symbolic identifier. This highlights a critical "attention drift," where standard coding conventions (lowercase variables) override local context (uppercase constants).

5.1.4 Namespace Collisions

A subtler but equally fatal failure mode involves Namespace Shadowing, in which the model chooses variable names that clash with imported libraries. This behavior indicates a lack of whole-program awareness, as the model effectively overwrites its own tools.

Listing 5.3: Hallucination of a Non-Existent Global Function `random_seed`

```
import random
import time
import matplotlib.pyplot as plt

def task_func(my_list, size=100, seed=100):
    if not isinstance(my_list, list):
        raise TypeError("my_list must be a list")
    if not all(isinstance(x, (int, float)) for x in my_list):
        raise ValueError("my_list must contain only numeric elements")

    random_seed(seed)
    start_time = time.time()

    my_list.append(12)
    total_sum = sum(my_list)
    random_list = [random.randint(1, min(100, total_sum)) for _ in range(size)]

    end_time = time.time()
    time_taken = end_time - start_time

    plt.hist(random_list, bins=range(1, 102), edgecolor='black')
    plt.xlabel('Number')
    plt.ylabel('Frequency')
    plt.title('Histogram of Random Numbers')

    return time_taken, plt.gcf().axes[0]
```

Listing 5.5 demonstrates a representative instance of this logic failure. The model begins by importing the `stats` module from `scipy`. However, when the task requires calculating summary metrics, the model defines a local dictionary, `stats`, to store the mean and standard deviation.

While `stats` is a semantically logical name for this dictionary, it destructively shadows the imported module. When the model subsequently attempts to access `stats.norm.fit(arr)`, the interpreter attempts to look up the `.norm` attribute on the local dictionary rather than the module, resulting in a crash.

5.2 RQ2: Prompt Engineering Mechanisms

Our quantitative analysis in RQ2 indicated that prompt engineering significantly alters code structure, with Quality-Focused prompts improving quality while RCI degrades it. To understand the mechanisms behind these changes, we conducted a comparative analysis of code generated by the Zero-Shot Baseline, Quality-Focused, and RCI prompts.

Listing 5.4: Case sensitivity failure: The Model Defines `NUMBERS` but Later References the Undefined Lowercase `numbers`

```
import numpy as np
import matplotlib.pyplot as plt
import random

NUMBERS = list(range(1, 7))

def task_func(rolls, seed=None):
    if seed is not None:
        random.seed(seed)
    results = [random.choice(NUMBERS) for _ in range(rolls)]
    unique, counts = np.unique(results, return_counts=True)
    frequencies = np.asarray(counts) / rolls
    plt.hist(numbers, weights=frequencies, bins=len(NUMBERS))
    plt.title("Histogram of Dice Rolls")
    plt.xlabel("Dice Value")
    plt.ylabel("Frequency")
    plt.show()
    return frequencies, plt.gca()
```

5.2.1 Verbosity Trade-Offs

The RCI technique often increased smell density. A qualitative inspection reveals that this is rarely due to the introduction of incorrect logic, but rather an over-correction.

Listing 5.6 illustrates a representative transformation from the CoderEval benchmark. In the Zero-Shot solution, the model generates a concise, functional HTTP request. The RCI iteration, which attempts to critique and improve the solution, wraps the logic in an exhaustive `try-except` block that catches four distinct exception types and explicitly checks for a missing library.

While this refactored code is objectively more robust—notably fixing the *B113: Request Without Timeout* security smell by adding a `timeout=10` parameter—it introduces significant verbosity. This highlights a critical trade-off: LLM self-correction tends to conflate quality with handling every edge case, resulting in code that is secure but harder to maintain due to visual clutter.

5.2.2 Hallucinated Refactoring

A second failure mode of complex prompting is "Hallucinated Refactoring." When instructed to critique code, models often restructure logic into helper methods or introduce constants that were never defined.

In the validation example shown in Listing 5.7, the RCI prompt successfully breaks a monolithic `validate` function into modular helper methods. However, it introduces usage of `SPEC_VERSION_V1` and `SPEC_VERSION_V2`—constants that appear nowhere in the generated

Listing 5.5: Namespace Collision: The Local Variable `stats` Shadows the Imported `scipy.stats` Module

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

def task_func(original):
    arr = np.array([t[1] for t in original])
    stats = {
        'mean': np.mean(arr),
        'std': np.std(arr),
        'min': np.min(arr),
        'max': np.max(arr)
    }
    mu, sigma = stats.norm.fit(arr)
    n, bins, patches = plt.hist(arr, density=True, alpha=0.6, bins='auto', label='
        Histogram')
    y = stats.norm.pdf(bins, mu, sigma)
    plt.plot(bins, y, 'r--', linewidth=2, label='PDF')
    plt.legend()
    ax = plt.gca()
    return arr, stats, ax
```

scope. Unlike the Zero-Shot generation, which used literal strings ('v1', 'v2'), this improved code is syntactically elegant but functionally broken.

It remains an open question for future work whether this degradation stems from the model's inability to correctly identify smells during the critique phase (false positives in self-assessment) or an inability to execute the refactoring without introducing new errors. However, the result—hallucinated refactoring—suggests that the 'fix' phase is particularly prone to over-correction.

5.2.3 Implicit Code Hygiene

In contrast to the structural correction of RCI, the Quality-Focused prompt demonstrated a capacity for subtle yet effective "Modernization and Hygiene."

Listing 5.8 illustrates a dual improvement in a Pandas data processing task. First, regarding API stability, Zero-Shot utilized `pd.api.types.is_categorical_dtype`, a method that has faced deprecation warnings in recent library versions. The Quality-Focused model replaced this with the more stable numpy-based check `np.issubdtype` without explicit instruction.

Second, regarding code hygiene, the Zero-Shot generated a significant smell by unpacking all four return values from `chi2_contingency` (`chi2, p, dof, expected`), even though it only returned `p`. This triggers the *W0612: Unused Variable* warning three times. The Quality-Focused solution adopted the Python convention of using underscores (_) to discard unused values explicitly. This confirms that simple quality constraints effectively activate the model's

latent knowledge of both library evolution and clean coding standards, resolving multiple smell categories (*Warning* and *Convention*) simultaneously.

5.3 Threats to Validity

Empirical software engineering research involves inherent limitations that constrain the generalizability and interpretation of findings. Below, we discuss threats to validity for our study and describe how we aimed to mitigate them.

5.3.1 Construct Validity

Reliability of Static Analysis We relied on automated static analysis tools (Pylint and Bandit) to quantify code smells. While these tools are industry standards, they function based on heuristics rather than semantic understanding. A significant threat to our findings is classifying missing context as an error. As observed in the CoderEval benchmark, Pylint frequently flagged *E0602: Undefined variable* errors because it analyzed functions in isolation, lacking access to the broader project scope.

Suppression of Docstrings and Comments All models were explicitly instructed to avoid generating docstrings and comments Section 3.2.3, a decision made to prevent excessive verbosity and reduce the risk of truncated responses due to token limits during preliminary trials. While this ensured greater consistency in output length and maintained focus on executable logic, it introduces a non-trivial threat to construct validity. In practical software development, docstrings and comments contribute significantly to readability and maintainability. Several static analysis heuristics—particularly within Pylint—are influenced by the presence or absence of such documentation. By suppressing these natural language elements, the study intentionally excluded documentation-related smells from the analysis, potentially underrepresenting an important quality dimension that would typically be present in real-world code. Therefore, the reported results should be interpreted primarily as a measure of the internal structural quality of bare, uncommented code, rather than as a comprehensive evaluation of full, production-ready software artifacts.

5.3.2 Internal Validity

Prompt Sensitivity Our evaluation of prompt engineering techniques relied on specific, fixed templates. LLMs are known to be highly sensitive to minor phrasing variations. The negative performance of the RCI technique may stem from the specific wording of our critique instruction rather than an inherent failure of self-correction mechanisms. Therefore, our conclusions regarding prompting efficacy apply strictly to the specific templates used in this study.

Data Contamination A prevalent risk in evaluating LLMs is data contamination. While we selected BigCodeBench-Hard specifically for its recent release to minimize this risk, we cannot guarantee that the open-source models (Phi-3-mini, Phi-4, Qwen2.5-Coder) have not been exposed to similar problems or earlier versions of the CoderEval dataset.

5.3.3 External Validity

Model Selection We restricted our analysis to three open-source models. This selection excludes proprietary state-of-the-art models (e.g., GPT-5.1, Gemini 3), which may possess superior instruction-following capabilities that could alter the effectiveness of prompting techniques like RCI.

Programming Language Bias This study focused exclusively on Python. Python's dynamic typing and whitespace-dependent syntax create specific categories of smells that do not exist in statically typed languages. As such, specific smell profiles—particularly indentation errors—are language-specific artifacts. Future work must extend this methodology to other languages to determine if these quality trends remain consistent.

Listing 5.6: Comparison of Zero-Shot vs. RCI: RCI Corrects the Security Timeout Smell but Introduces Excessive Boilerplate

```
# --- Zero-Shot Generation ---
def retrieve_diaspora_host_meta(host):
    import requests
    from xrd import XRD
    response = requests.get(f'https://{host}/.well-known/host-meta')
    if response.status_code == 200:
        return XRD(response.text)
    else:
        raise Exception(f"Failed to retrieve host-meta: {response.status_code}")

# --- RCI Generation ---
def retrieve_diaspora_host_meta(host):
    import requests
    try:
        from xrd import XRD
    except ImportError:
        raise ImportError("The 'xrd' library is required but not installed. Please
            install it using pip.")

    url = host if host.startswith(('http://', 'https://')) else f'https://{host}'
    try:
        response = requests.get(url + '/.well-known/host-meta', timeout=10)
        response.raise_for_status()
        return XRD(response.text)
    except requests.exceptions.HTTPError as e:
        raise Exception(f"HTTP error occurred: {e}")
    except requests.exceptions.ConnectionError as e:
        raise Exception(f"Connection error occurred: {e}")
    except requests.exceptions.Timeout as e:
        raise Exception(f"Timeout error occurred: {e}")
    except requests.exceptions.RequestException as e:
        raise Exception(f"Request error occurred: {e}")
    except Exception as e:
        raise Exception(f"Failed to parse XRD: {e}")
```

Listing 5.7: Hallucinated Refactoring: RCI Introduces Undefined Constants (`SPEC_VERSION_V1`) in an Attempt to Make the Code Look More Modular

```
# --- Zero-Shot Generation ---
def validate(self, inventory, extract_spec_version=False):
    if extract_spec_version:
        spec_version = inventory.get('type', None)
        if spec_version not in ['v1', 'v2']:
            spec_version = self.spec_version
    else:
        spec_version = self.spec_version

    # Placeholder for validation logic based on spec_version
    if spec_version == 'v1':
        # Validation logic for v1
        pass
    elif spec_version == 'v2':
        # Validation logic for v2
        pass
    else:
        raise ValueError("Invalid specification version")

# --- RCI Generation ---
def validate(self, inventory, extract_spec_version=False):
    if extract_spec_version:
        spec_version = inventory.get('type', None)
        if spec_version not in [SPEC_VERSION_V1, SPEC_VERSION_V2]:
            spec_version = self.spec_version
    else:
        spec_version = self.spec_version

    if spec_version == SPEC_VERSION_V1:
        return self._validate_v1(inventory)
    elif spec_version == SPEC_VERSION_V2:
        return self._validate_v2(inventory)
    else:
        raise ValueError(f"Unsupported specification version: {spec_version}")

def _validate_v1(self, inventory):
    # Validation logic for v1
    return True  # Placeholder return value

def _validate_v2(self, inventory):
    # Validation logic for v2
    return True  # Placeholder return value
```

Listing 5.8: The Quality-Focused Prompt (Bottom) Simultaneously Modernizes Deprecated APIs and Eliminates Unused Variables Present in the Zero-Shot (Top)

```
# --- Zero-Shot Generation ---
import pandas as pd
import numpy as np
from scipy.stats import chi2_contingency

def task_func(data, col1, col2):
    if data.empty:
        raise ValueError("Data is empty.")
    if col1 not in data.columns or col2 not in data.columns:
        raise ValueError(f"Columns {col1} and/or {col2} not found in data.")
    if not pd.api.types.is_categorical_dtype(data[col1]) or not pd.api.types.
        is_categorical_dtype(data[col2]):
        raise TypeError(f"Columns {col1} and {col2} must be categorical.")
    if len(data[col1].unique()) < 2 or len(data[col2].unique()) < 2:
        raise ValueError(f"Columns {col1} and {col2} must have at least two categories
            .")

    contingency_table = pd.crosstab(data[col1], data[col2])
    if (contingency_table.values < 5).any():
        raise ValueError("Some categories have less than 5 observations, violating chi
            -square test assumptions.")

    chi2, p, dof, expected = chi2_contingency(contingency_table)
    return p

# --- Quality-Focused Generation ---
import pandas as pd
import numpy as np
from scipy.stats import chi2_contingency

def task_func(data, col1, col2):
    if data.empty:
        raise ValueError("Dataframe is empty.")
    if col1 not in data.columns or col2 not in data.columns:
        raise ValueError(f"Columns {col1} and/or {col2} not found in dataframe.")
    if not all(np.issubdtype(dtype, np.object_) for dtype in [data[col1].dtype, data[
        col2].dtype]):
        raise TypeError(f"Columns {col1} and {col2} must be categorical.")
    if len(data[col1].unique()) < 2 or len(data[col2].unique()) < 2:
        raise ValueError(f"Columns {col1} and {col2} must have at least two categories
            each.")

    contingency_table = pd.crosstab(data[col1], data[col2])
    if (contingency_table.values < 5).any():
        raise ValueError("Contingency table contains cells with less than 5
            observations.")

    _, p_value, _, _ = chi2_contingency(contingency_table)
    return p_value
```

6

Related Work

This chapter situates our research within the broader academic discourse on LLM-based software engineering. We organize the related work around three primary themes: the structural quality of code generated by LLMs (RQ1), the impact of prompt engineering on generation quality (RQ2), and the evolution of evaluation benchmarks that enable systematic comparison.

6.1 LLM-Generated Code Quality

The empirical investigation of code quality in the output of LLMs has recently emerged as a critical research direction. While early studies focused primarily on functional correctness, recent work has begun to scrutinize the maintainability and structural integrity of generated solutions.

Paul et al. [59] conducted a comprehensive scenario-based evaluation that compared code smells in solutions generated by four state-of-the-art LLMs (Gemini Pro, ChatGPT, Codex, and Falcon) against reference implementations written by humans. Their study found that across the evaluated models, the incidence of code smells increased by an average of 63.34%. Specifically, implementation-level smells accounted for 73.35% of the detected issues, while design-level smells accounted for 21.42%. Critically, they observed that the problem's complexity correlates more strongly with the propensity for smells than with the model selection. This suggests that task characteristics significantly influence quality outcomes. While their work establishes a critical baseline for the prevalence of smells in LLM output, it does not investigate whether strategic prompt engineering interventions can reduce the generation of these smells—a gap our work directly addresses.

In a parallel empirical study, researchers evaluated the maintainability and reliability of Python code generated by LLMs using SonarQube static analysis [53]. Their evaluation across three difficulty levels (introductory, interview, and competition) revealed that while LLM-generated code sometimes exhibits fewer runtime bugs overall, it introduces distinctive structural issues at higher levels of complexity. This suggests a potential decline in quality as task sophistication increases, motivating our investigation into whether prompt engineering can mitigate this structural degradation across different task complexities.

Furthermore, Alves et al. [5] specifically investigated the quality of test code from GitHub Copilot. They found that 47.4% of Python test cases generated by Copilot contained test smells, even though they were functionally correct. This parallel finding in the test code domain reinforces the importance of evaluating internal quality dimensions beyond simple functional correctness—a core motivation for our systematic smell analysis.

6.2 Static Analysis Tools

Understanding which quality assessment tools practitioners trust and rely on informs the methodological choices in empirical research. A recent empirical study combining interviews with 10 software development professionals and surveys of 310 practitioners [88] revealed a significant gap between research contributions and practical needs regarding code smell detection tools. Practitioners prioritize actionable recommendations over mere detection and expressed skepticism toward approaches driven purely by machine learning.

Their study analyzed 78 papers on code smell detection published between 2020 and 2024, revealing a trend toward machine learning in academia, while practitioners continue to rely heavily on traditional static analysis tools such as SonarQube and Pylint. This preference for established, interpretable tools justifies our methodological choice of Pylint and Bandit as the evaluation apparatus for this thesis. These tools provide deterministic, rule-based detection that produces actionable, categorized feedback—precisely what practitioners value in production environments.

6.3 Prompt Engineering Techniques

The broader field of prompt engineering has established foundational techniques for guiding the behavior of LLMs. However, a systematic investigation of how prompting strategies influence code quality—specifically internal attributes such as code smells—remains absent.

Zero-Shot, Few-shot, and CoT prompting represent established techniques in prompt engineering. Despite their widespread adoption, Porta et al. [62] conducted an empirical assessment using the Dev-GPT dataset with 7,583 code files across three quality metrics (maintainability, security, and reliability). Remarkably, their statistical testing revealed no significant differences in the impact of prompts on code quality metrics. This suggests that the prompt structure may not substantially influence these quality dimensions in code generated by ChatGPT. This finding contradicts assumptions in the prompt engineering literature and motivates our investigation into whether different quality attributes—specifically code smells—respond differently to prompt interventions in open-source models.

Khojah et al. [39] introduced CodePromptEval, a dataset of 7,072 prompts across five prompt techniques and three LLMs, enabling systematic evaluation of prompt effectiveness. Their central finding—that *combined techniques do not necessarily improve outcomes* and that *significant trade-offs exist between correctness and quality*. They observed that CoT prompting sometimes reduces functional correctness while potentially improving other quality attributes. This establishes that the effects of prompting on code characteristics are non-uniform and potentially conflicting, motivating our approach of analyzing internal quality above functional correctness.

The interaction between the prompt design and the model's behavior emerged from research on the energy efficiency of generated code [7]. When evaluating small language models across four prompting strategies (role prompting, Zero-Shot, Few-Shot, and CoT), researchers observed that CoT consistently improved energy for some models but failed

for others. This indicates that prompting effects depend heavily on model architecture—a hypothesis we test across three distinct models.

Huang et al. [30] conducted a bias mitigation study in code generated by LLMs, employing Zero-Shot, One-Shot, and Few-Shot variants alongside CoT approaches. Their key finding indicates that direct prompt engineering is ineffective at mitigating specific code characteristics, whereas test-driven feedback substantially improves outcomes. This suggests that combining prompting strategies with external feedback loops yields more substantial quality improvements than prompting alone. Our investigation of RCI tests whether internal feedback loops (model critiquing its own output) can approximate the benefits of external feedback without requiring test suites.

6.4 Code Generation Benchmarks

Our empirical methodology relies on established code generation benchmarks that provide standardized evaluation environments. The benchmark landscape has historically been dominated by HumanEval [12] (164 Python problems) and MBPP [8] (974 primarily introductory problems). These provide standardized test suites for evaluating functional correctness and enable reproducible evaluation, but they historically focus exclusively on correctness metrics.

However, critical examination of benchmark quality itself has recently emerged as a research priority. Siddiq et al. [68] conducted the first comprehensive study of task quality in code-generation benchmarks, analyzing 3,566 tasks across nine datasets. Their findings revealed that benchmarks contain spelling errors, grammatical issues, unclear intent expressions, and inconsistent documentation. These systematic quality issues potentially mislead performance comparisons and introduce confounding variables. This motivates careful validation of benchmark task descriptions before experimental execution, as reflected in our methodology, which uses the curated BigCodeBench-Hard dataset [68] to address these quality concerns explicitly.

Recent extensions like HumanEval Pro and MBPP Pro [87] introduce self-invoking code-generation tasks in which models must compose solutions from outputs of simpler problems. This represents an evolution toward more complex, compositional evaluation paradigms. Our work focuses on traditional, isolated function generation, leveraging established baseline benchmarks while acknowledging emerging evaluation directions. This choice enables direct comparison with the substantial body of prior work while providing a controlled environment for isolating the prevalence of smell.

7

Concluding Remarks

This thesis systematically evaluated the internal quality of code generated by open-source LLMs. While prior research extensively evaluated functional correctness, the maintainability of LLM-generated code—measured through code smells—remained an underexplored area. By analyzing over 6,000 code samples across three models (Phi-3-mini, Phi-4, Qwen2.5-Coder), five prompting strategies, and two realistic benchmarks (CoderEval, BigCodeBench-Hard), we provided a systematic assessment of code smell prevalence in open-source models and the efficacy of prompt engineering interventions for quality improvement. This chapter synthesizes our contributions and outlines directions for future research.

7.1 Contributions

This thesis makes three primary contributions to the field of LLM-assisted software engineering:

1. **Empirical Baseline for Open-Source Models**: We established a systematic quantification of code smell prevalence in open-source LLMs, analyzing smell density and distribution across different categories. We employed industry-standard static analysis tools (Pylint and Bandit) to enable reproducible comparisons.

2. **Human-LLM Quality Comparison**: We conducted a direct comparison between LLM-generated code and human-written canonical solutions from the same benchmarks, revealing fundamentally different quality profiles. This comparison demonstrates that while LLMs produce fewer total smells, they exhibit distinct failure modes (structural/-logical errors) compared to human developers (stylistic violations).

3. **Systematic Prompt Engineering Evaluation**: We evaluated five distinct prompting techniques across multiple models and benchmarks, revealing that simple constraint-based prompts outperform complex recursive self-correction methods. This finding challenges assumptions in the prompt engineering literature about the efficacy of sophisticated multi-turn reasoning strategies for improving code quality.

7.2 Future Directions

The findings of this thesis open several avenues for further investigation into the intersection of Generative AI and software quality.

7.2.1 Extended Validation

Cross-Language Generalization This study focused exclusively on Python, where whitespace sensitivity creates unique formatting smells (e.g., *W0311: Bad indentation*). Future work should extend this methodology to statically typed languages such as Java or C++. Investigating whether the Syntax-Logic Gap persists in languages with strict compile-time checks would help isolate language-specific artifacts from inherent model limitations. Additionally, analyzing functional languages (Haskell, Scala) would reveal whether type systems mitigate the namespace confusion observed in dynamic languages.

Repository-Aware Generation We found that *E0602: Undefined variable* were a major cause of code smells. This often occurs because models analyze functions in isolation and hallucinate variables expected to exist in a broader project context. Real software development happens in large repositories, not isolated snippets. Future research should investigate repository-aware generation using Retrieval-Augmented Generation (RAG), where the model accesses other files in the project to resolve correct variable names, imports, and API signatures. This could directly address the Syntax-Logic Gap by providing semantic grounding.

7.2.2 Tool Integration

Static Analysis Feedback Loops In this study, we used Pylint only after code generation to measure quality. A more realistic approach integrates static analysis into the generation process. Since we found that models struggle to self-correct through prompts alone (the failure of RCI), future work should build agentic workflows where the model generates code, immediately runs a linter, observes concrete error messages, and iteratively fixes issues before presenting results to users. This combines the creativity of LLMs with the reliability of deterministic verification tools, potentially surpassing both pure prompting and pure post-processing approaches.

Closing Remarks

As LLMs become increasingly integrated in the software development lifecycle, the focus must shift from merely generating code that runs to generating code that lasts. This thesis provides a foundational step toward that goal, establishing that internal quality is a measurable, distinct, and manageable dimension of LLM-assisted engineering. Moving forward, we need to build integrated workflows that combine the generative creativity of LLMs with the verification rigor of traditional software-engineering tools. This approach allows us to use LLM to produce code that is not only functionally correct, but also maintainable in the long run.

A

Appendix

Table A.1: Complete Code Smell Summary for Phi-3-mini on BigCodeBench - Hard Across All Five Prompting Techniques Categorized by Message Type

Prompt	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly (%)	Density
Zero-Shot	51	21	144	39	70	325	74.13	2.27
Quality-Focused	63	19	141	44	76	343	80.69	2.37
Persona-Based	59	19	167	68	69	382	76.03	2.62
CoT	54	18	160	26	63	321	79.58	2.26
RCI	69	21	203	32	61	386	83.46	2.90

Table A.2: Complete Code Smell Summary for Phi-3-mini on CoderEval Across All Five Prompting Techniques Categorized by Message Type

Prompt	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly (%)	Density
Zero-Shot	50	98	104	87	28	367	62.96	1.70
Quality-Focused	41	89	80	72	31	313	57.66	1.41
Persona-Based	48	75	137	66	43	369	58.04	1.65
CoT	50	111	115	51	38	365	66.22	1.62
RCI	82	90	144	62	33	411	68.49	1.88

Table A.3: Complete Code Smell Summary for Phi-4 on BigCodeBench-Hard Across All Five Prompting Techniques Categorized by Message Type

Prompt	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly (%)	Density
Zero-Shot	47	16	121	16	70	270	72.97	1.82
Quality-Focused	39	16	124	31	67	277	72.11	1.88
Persona-Based	45	19	127	25	66	282	75.51	1.92
CoT	52	17	135	22	66	292	75.00	1.97
RCI	73	27	161	22	70	353	82.43	2.39

Table A.4: Complete Code Smell Summary for Phi-4 on CoderEval Across All Five Prompting Techniques Categorized by Message Type

Prompt	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly (%)	Density
Zero-Shot	32	96	88	20	35	271	52.40	1.18
Quality-Focused	41	90	81	7	31	250	52.17	1.09
Persona-Based	43	75	105	15	31	269	53.48	1.17
CoT	32	97	102	6	30	267	54.35	1.16
RCI	50	90	129	40	36	345	61.74	1.50

Table A.5: Complete Code Smell Summary for Qwen2.5-Coder on BigCodeBench-Hard Across All Five Prompting Techniques Categorized by Message Type

Prompt	#Conv	#Ref	#Warn	#Err	#Sec	#Total	Smelly (%)	Density
Zero-Shot	50	19	122	21	66	278	78.38	1.88
Quality-Focused	43	21	119	24	65	272	75.68	1.84
Persona-Based	57	19	122	27	61	286	80.95	1.95
CoT	45	21	112	18	64	260	76.35	1.76
RCI	86	36	176	20	65	383	88.44	2.61

Table A.6: Complete Code Smell Summary for Qwen2.5-Coder on CoderEval Across All Five Prompting Techniques Categorized by Message Type

Prompt	**#Conv**	**#Ref**	**#Warn**	**#Err**	**#Sec**	**#Total**	**Smelly (%)**	**Density**
Zero-Shot	34	37	204	91	30	396	53.95	1.74
Quality-Focused	23	24	84	67	32	230	41.74	1.00
Persona-Based	46	36	150	78	40	350	51.74	1.52
CoT	29	31	95	63	31	249	46.93	1.09
RCI	56	40	164	73	32	365	56.14	1.60

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used:

We have used GitHub Copilot for code completion and Gemini for inspiration on how to visualise the data. In addition, we have used ChatGPT to convert CSV tables to LaTeX and to fix LaTeX formatting issues. We have also used ChatGPT to search for related literature. We have used Claude to check grammar and formatting and to refine the text toward an academic style.

We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

Bibliography

[1] Altaf Allah Abbassi, Leuson Da Silva, Amin Nikanjam, and Foutse Khomh. "A Taxonomy of Inefficiencies in LLM-Generated Python Code." In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2025), pp. 393–404.

[2] Marah Abdin et al. *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*. 2024. arXiv: 2404.14219 [cs.CL]. URL: https://arxiv.org/abs/2404.14219.

[3] Marah Abdin et al. *Phi-4 Technical Report*. 2024. arXiv: 2412.08905 [cs.CL]. URL: https://arxiv.org/abs/2412.08905.

[4] Christopher Akiki, Giada Pistilli, Margot Mieskes, Matthias Gallé, Thomas Wolf, Suzana Ilic, and Yacine Jernite. "BigScience: A Case Study in the Social Construction of a Multilingual Large Language Model." In: *Workshop on Broadening Research Collaborations 2022*. 2022.

[5] Victor Alves, Cristiano Santos, Carla Bezerra, and Ivan Machado. "Detecting Test Smells in Python Test Code Generated by LLM: An Empirical Study With GitHub Copilot." In: 2024, pp. 581–587.

[6] Elham Asgari, Nina Montaña-Brown, Magda Dubois, Saleh Khalil, Jasmine Balloch, Joshua Au Yeung, and Dominic Pimenta. "A Framework to Assess Clinical Safety and Hallucination Rates of LLMs for Medical Text Summarization." In: *npj Digital Medicine* 8.1 (2025), p. 274.

[7] Humza Ashraf, Syed Muhammad Danish, Shadikur Rahman, and Zeeshan Sattar. *Toward Green Code: Prompting Small Language Models for Energy-Efficient Code Generation*. 2025. arXiv: 2509.09947 [cs.SE]. URL: https://arxiv.org/abs/2509.09947.

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL]. URL: https://arxiv.org/abs/2108.07732.

[9] Brenton T Bicknell, Danner Butler, Sydney Whalen, James Ricks, Cory J Dixon, Abigail B Clark, Olivia Spaedy, Adam Skelton, Neel Edupuganti, Lance Dzubinski, Hudson Tate, Garrett Dyess, Brenessa Lindeman, and Lisa Soleymani Lehmann. "ChatGPT-4 Omni Performance in USMLE Disciplines and Clinical Skills: Comparative Analysis." In: *JMIR Med Educ* 10 (2024), e63430.

[10] Tom Brown et al. "Language Models Are Few-Shot Learners." In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2020, pp. 1877–1901.

[11] Sivajeet Chand, Melih Kilic, Roland Würsching, Sushant Kumar Pandey, and Alexander Pretschner. *Automated Extract Method Refactoring with Open-Source LLMs: A Comparative Study*. 2025. arXiv: 2510.26480 [cs.SE]. URL: https://arxiv.org/abs/2510.26480.

[12] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[13] Xinyun Chen, Ryan A. Chi, Xuezhi Wang, and Denny Zhou. *Premise Order Matters in Reasoning with Large Language Models*. 2024. arXiv: 2402.08939 [cs.AI]. URL: https://arxiv.org/abs/2402.08939.

[14] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. "Detecting Code Smells in Python Programs." In: *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 2016, pp. 18–23.

[15] Aakanksha Chowdhery et al. "PaLM: scaling language modeling with pathways." In: *Journal of Machine Learning Research (JMLR)* 24.1 (2023).

[16] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[17] Bandit Developers. *Bandit*. Website. Available online at https://bandit.readthedocs.io/en/latest/index.html; visited on August 1st, 2025. 2025.

[18] Marc Dillmann, Julien Siebert, and Adam Trendowicz. *Evaluation of large language models for assessing code maintainability*. 2024. arXiv: 2401.12714 [cs.SE]. URL: https://arxiv.org/abs/2401.12714.

[19] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[20] Martin Fowler, K Beck, J Brant, W Opdyke, and D Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[21] Isabel O. Gallegos, Ryan A. Rossi, Joe Barrow, Md Mehrab Tanjim, Sungchul Kim, Franck Dernoncourt, Tong Yu, Ruiyi Zhang, and Nesreen K. Ahmed. "Bias and Fairness in Large Language Models: A Survey." In: *Computational Linguistics* 50.3 (2024), pp. 1097–1179.

[22] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*. 2020. arXiv: 2101.00027 [cs.CL]. URL: https://arxiv.org/abs/2101.00027.

[23] GitHub. *Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness*. Website. Available online at https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/; visited on July 30th, 2025. 2022.

[24] GitHub. *GitHub Copilot. Your AI pair programmer*. Website. Available online at https://copilot.github.com/; visited on July 31th, 2025. 2023.

[25] Van Rossum Guido, Warsaw Barry, and Coghlan Alyssa. *PEP 8 – Style Guide for Python Code*. Website. Available online at https://peps.python.org/pep-0008/; visited on August 1st, 2025. 2001.

[26] Zhiguang Han and Zijian Wang. "Rethinking the Role-Play Prompting in Mathematical Reasoning Tasks." In: *Proceedings of the 1st Workshop on Efficiency, Security, and Generalization of Multimedia Foundation Models*. Association for Computing Machinery, 2024, pp. 13–17.

[27] Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. *Do Large Code Models Understand Programming Concepts? Counterfactual Analysis for Code Predicates*. 2025. arXiv: 2402.05980 [cs.SE]. URL: https://arxiv.org/abs/2402.05980.

[28] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "LoRA: Low-Rank Adaptation of Large Language Models." In: *International Conference on Learning Representations*. 2022.

[29] Huimin Hu, Yingying Wang, Julia Rubin, and Michael Pradel. "An Empirical Study of Suppressed Static Analysis Warnings." In: *Proc. ACM Softw. Eng.* 2.FSE (2025), FSE014.

[30] Dong Huang, Jie M Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. "Bias Testing and Mitigation in LLM-Based Code Generation." In: *ACM Transactions on Software Engineering and Methodology* (2024).

[31] Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, and Weizhu Chen. "Competition-Level Problems are Effective LLM Evaluators." In: *Findings of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2024, pp. 13526–13544.

[32] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. 2020. arXiv: 1909.09436 [cs.LG]. URL: https://arxiv.org/abs/1909.09436.

[33] Che Jiang, Biqing Qi, Xiangyu Hong, Dayuan Fu, Yang Cheng, Fandong Meng, Mo Yu, Bowen Zhou, and Jie Zhou. "On Large Language Models' Hallucination with Regard to Known Facts." In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Association for Computational Linguistics, 2024, pp. 1041–1053.

[34] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. *Self-planning Code Generation with Large Language Models*. 2024. arXiv: 2303.06689 [cs.SE]. URL: https://arxiv.org/abs/2303.06689.

[35] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. "Highly Accurate Protein Structure Prediction with AlphaFold." In: *Nature* 596.7873 (2021), pp. 583–589.

[36] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[37] Daniel Martin Katz, Michael James Bommarito, Shang Gao, and Pablo Arredondo. "GPT-4 Passes the Bar Exam." In: *Philosophical Transactions of the Royal Society A* 382.2270 (2024), p. 20230254.

[38] Zena Al Khalili, Nick Howell, and Dietrich Klakow. "Evaluating Intermediate Reasoning of Code-Assisted Large Language Models for Mathematics." In: *Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM²)*. Association for Computational Linguistics, 2025, pp. 741–758.

[39] Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. *The Impact of Prompt Programming on Function-Level Code Generation*. 2025. arXiv: 2412.20545 [cs.SE]. URL: https://arxiv.org/abs/2412.20545.

[40] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness." In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[41] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. "Language Models Can Solve Computer Tasks." In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2023, pp. 39648–39677.

[42] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. "Better Zero-Shot Reasoning with Role-Play Prompting." In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*. Association for Computational Linguistics, 2024, pp. 4099–4113.

[43] Arjun Krishna, Erick Galinkin, Leon Derczynski, and Jeffrey Martin. *Importing Phantoms: Measuring LLM Package Hallucination Vulnerabilities*. 2025. arXiv: 2501.19012 [cs.LG]. URL: https://arxiv.org/abs/2501.19012.

[44] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models." In: *Proceedings of the 45th International Conference on Software Engineering*. IEEE Press, 2023, pp. 919–931.

[45] Moxin Li, Yong Zhao, Wenxuan Zhang, Shuaiyi Li, Wenya Xie, See-Kiong Ng, Tat-Seng Chua, and Yang Deng. "Knowledge Boundary of Large Language Models: A Survey." In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2025, pp. 5131–5157.

[46] Raymond Li et al. "StarCoder: May the Source Be With You!" In: *Transactions on Machine Learning Research* (2023).

[47] Yujia Li et al. "Competition-level code generation with AlphaCode." In: *Science* 378.6624 (2022), pp. 1092–1097.

[48] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation." In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2023, pp. 21558–21572.

[49] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. "Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing." In: *ACM Computing Surveys* 55 (2021), pp. 1–35.

[50] Logilab and Pylint contributors. *Pylint*. Website. Available online at https://pylint.pycqa.org/en/latest/index.html; visited on August 1st, 2025. 2025.

[51] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. "Self-Refine: Iterative Refinement with Self-Feedback." In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023, pp. 46534–46594.

[52] Jiya Manchanda, Laura Boettcher, Matheus Westphalen, and Jasser Jasser. *The Open Source Advantage in Large Language Models (LLMs)*. 2025. arXiv: 2412.12004 [cs.CL]. URL: https://arxiv.org/abs/2412.12004.

[53] Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assuncao. *Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code?* 2025. arXiv: 2508.00700 [cs.SE]. URL: https://arxiv.org/abs/2508.00700.

[54] Thu-Trang Nguyen, Thanh Trong Vu, Hieu Dinh Vo, and Son Nguyen. "An Empirical Study on Capability of Large Language Models in Understanding Code Semantics." In: *Information and Software Technology* 185 (2025), p. 107780.

[55] Bart van Oort, Luís Cruz, Maurício Aniche, and Arie van Deursen. "The Prevalence of Code Smells in Machine Learning Projects." In: *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. 2021, pp. 1–8.

[56] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. "Training Language Models to Follow Instructions with Human Feedback." In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2022, pp. 27730–27744.

[57] Stack Overflow. *Stack Overflow Developer Survey 2024*. Website. Available online at https://survey.stackoverflow.co/2024/; visited on July 30th, 2025. 2024.

[58] PMD. *PMD Source Code Analyzer*. Website. Available online at https://pmd.github.io; visited on November 19th, 2025. 2025.

[59] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. *Investigating The Smells of LLM Generated Code*. 2025. arXiv: 2510.03029 [cs.SE]. URL: https://arxiv.org/abs/2510.03029.

[60] Fabiano Pecorelli, Savanna Lujan, Valentina Lenarduzzi, Fabio Palomba, and Andrea De Lucia. "On the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction." In: *Empirical Software Engineering (ESE)* 27.3 (2022), p. 64.

[61] Raluca Ada Popa and Four Flynn. *Introducing CodeMender: an AI agent for code security*. Website. Available online at https://deepmind.google/blog/introducing-codemender-an-ai-agent-for-code-security/; visited on Nov 16th, 2025. 2025.

[62] Antonio Della Porta, Stefano Lambiase, and Fabio Palomba. *Do Prompt Patterns Affect Code Quality? A First Empirical Assessment of ChatGPT-Generated Code*. 2025. arXiv: 2504.13656 [cs.SE]. URL: https://arxiv.org/abs/2504.13656.

[63] Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. *CodeElo: Benchmarking Competition-Level Code Generation of LLMs with Human-Comparable Elo Ratings*. 2025. arXiv: 2501.01257 [cs.CL]. URL: https://arxiv.org/abs/2501.01257.

[64] Hafiz Arslan Ramzan, Minahil Rauf, Sadia Ramzan, Yusra Zainab, Tehmina Kalsum, and Saba Iram. "Leveraging Large Language Models (LLMs) for Automated Bug Detection and Resolution in Software Engineering." In: 2025, pp. 1–6.

[65] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2025. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927.

[66] Rana Sandouka and Hamoud Aljamaan. "Python Code Smells Detection Using Conventional Machine Learning Models." In: *PeerJ Computer Science* 9 (2023), e1370.

[67] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. "Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I Learned to Start Worrying About Prompt Formatting." In: *International Conference on Representation Learning (ICLR)*. OpenReview.net, 2024, pp. 25055–25083.

[68] Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna C. S. Santos. "The Fault in Our Stars: Quality Assessment of Code Generation Benchmarks." In: *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. 2024, pp. 201–212.

[69] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. "An Empirical Study of Code Smells in Transformer-Based Code Generation Techniques." In: *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 71–82.

[70] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. "Quality Assessment of ChatGPT Generated Code and Their Use by Developers." In: *International Conference on Mining Software Repositories (MSR)*. Association for Computing Machinery, 2024, pp. 152–156.

[71] Sonar. *The Coding Personalities of Leading LLMs – A State of Code Report*. Website. Available online at https://www.sonarsource.com/the-coding-personalities-of-leading-llms.pdf; visited on November 17th, 2025. 2025.

[72] Da Song, Zijie Zhou, Zhijie Wang, Yuheng Huang, Shengmai Chen, Bonan Kou, Lei Ma, and Tianyi Zhang. "An Empirical Study of Code Generation Errors Made by Large Language Models." In: *7th Annual Symposium on Machine Programming*. 2023.

[73] Chitsutha Soomlek, Jan N. van Rijn, and Marcello M. Bonsangue. "Automatic Human-Like Detection of Code Smells." In: *Discovery Science: 24th International Conference, DS 2021, Halifax, NS, Canada, October 11–13, 2021, Proceedings*. Springer-Verlag, 2021, pp. 19–28.

[74] Joseph Spracklen, Raveen Wijewickrama, AHM Nazmus Sakib, Anindya Maiti, and Bimal Viswanath. "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs." In: *34th USENIX Security Symposium (USENIX Security 25)*. 2025, pp. 3687–3706.

[75] Niki van Stein, Anna V. Kononova, Lars Kotthoff, and Thomas Bäck. *Code Evolution Graphs: Understanding Large Language Model Driven Design of Algorithms*. 2025. arXiv: 2503.16668 [cs.NE]. URL: https://arxiv.org/abs/2503.16668.

[76] Pylint Development Team. *Pylint output - Pylint 4.0.4 documentation*. Website. Available online at https://pylint.readthedocs.io/en/stable/user_guide/usage/output.html; visited on December 1st, 2025. 2025.

[77] Qwen Team. *Qwen2.5: A Party of Foundation Models*. 2024. URL: https://qwenlm.github.io/blog/qwen2.5/.

[78] Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhif, and Riccardo Scandariato. "Prompting Techniques for Secure Code Generation: A Systematic Investigation." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2025).

[79] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and Removal of Type-Checking Bad Smells." In: *2008 12th European Conference on Software Maintenance and Reengineering*. 2008, pp. 329–331.

[80] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need." In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2017, pp. 6000–6010.

[81] Veracode. *GitHub Copilot. 2025 GenAI Code Security Report*. Website. Available online at https://www.veracode.com/wp-content/uploads/2025_GenAI_Code_Security_Report_Final.pdf; visited on November 17th, 2025. 2025.

[82] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. *Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval*. 2024. arXiv: 2407.02395 [cs.SE]. URL: https://arxiv.org/abs/2407.02395.

[83] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. "Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models." In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (2025), pp. 2587–2599.

[84] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." In: *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2022, pp. 24824–24837.

[85] Aiko Yamashita and Leon Moonen. "Do Code Smells Reflect Important Maintainability Aspects?" In: *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

[86] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models." In: *IEEE/ACM International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 2024, pp. 428–439.

[87] Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. "HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-Invoking Code Generation Task." In: *Findings of the Association for Computational Linguistics: ACL 2025*. Association for Computational Linguistics, 2025, pp. 13253–13279.

[88] Zexian Zhang, Shuang Yin, Wenliu Wei, Xiaoxue Ma, Jacky Wai Keung, Fuyang Li, and Wenhua Hu. "Practitioners' Expectations on Code Smell Detection." In: *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2024, pp. 1324–1333.

[89] Mingqian Zheng, Jiaxin Pei, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. "When "A Helpful Assistant" Is Not Really Helpful: Personas in System Prompts Do Not Improve Performances of Large Language Models." In: *Findings of the Association for Computational Linguistics: EMNLP 2024*. Association for Computational Linguistics, 2024, pp. 15126–15154.

[90] Jian-Qiao Zhu and Thomas L. Griffiths. *Incoherent Probability Judgments in Large Language Models*. 2025. arXiv: 2401.16646 [cs.CL]. URL: https://arxiv.org/abs/2401.16646.

[91] Terry Yue Zhuo et al. "BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions." In: *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2025, pp. 66602–66656.