

Are We SOLID Yet? An Empirical Study on Prompting LLMs to Detect Design Principle Violations

Fatih Pehlivan*, Arçin Ülkü Ergüzen*, Sahand Moslemi Yengejeh*, Mayasah Lami* and Anil Koyuncu†

Bilkent University, Turkey

* {fatih.pehlivan, ulku.erguzen, sahand.moslemi, m.lami}@bilkent.edu.tr

† anil.koyuncu@cs.bilkent.edu.tr

Abstract—Traditional static analysis methods struggle to detect semantic design flaws, such as violations of the SOLID principles, which require a strong understanding of object-oriented design patterns and principles. Existing solutions typically focus on individual SOLID principles or specific programming languages, leaving a gap in the ability to detect violations across all five principles in multi-language codebases. This paper presents a new approach: a methodology that leverages tailored prompt engineering to assess LLMs on their ability to detect SOLID violations across multiple languages. We present a benchmark of four leading LLMs—CodeLlama:70B, DeepSeekCoder:33B, Qwen2.5 Coder:32B, and GPT-4o Mini—on their ability to detect violations of all five SOLID principles. For this evaluation, we construct a new benchmark dataset of 240 manually validated code examples. Using this dataset, we test four distinct prompt strategies inspired by established zero-shot, few-shot, and chain-of-thought techniques to systematically measure their impact on detection accuracy. Our emerging results reveal a stark hierarchy among models, with GPT-4o Mini decisively outperforming others, yet even it struggles with challenging principles like DIP. Crucially, we show that prompt strategy has a dramatic impact, but no single strategy is universally best; for instance, a deliberative ENSEMBLE prompt excels at OCP detection while a hint-based EXAMPLE prompt is superior for DIP violations. Across all experiments, detection accuracy is heavily influenced by language characteristics and degrades sharply with increasing code complexity. These initial findings demonstrate that effective, AI-driven design analysis requires not a single “best” model, but a tailored approach that matches the right model and prompt to the specific design context, highlighting the potential of LLMs to support maintainability through AI-assisted code analysis.

Index Terms—SOLID Principles, Code Refactoring, Large Language Models, Prompt Patterns

I. INTRODUCTION

Ensuring high-quality, maintainable, and extensible software is a fundamental challenge in software engineering. While the SOLID principles—Single Responsibility (SRP), Open/Closed (OCP), Liskov Substitution (LSP), Interface Segregation (ISP), and Dependency Inversion (DIP)—provide a robust foundation for good design [1], violations are common and degrade code quality [2]. LLMs are now being integrated into developer workflows, but a critical, unaddressed question remains: Do these models understand the principles of good software design, or are they architecturally naive? An LLM that generates functionally correct but architecturally flawed code poses a significant long-term risk to software maintainability.

Existing analysis methods are insufficient to answer this question. Traditional methods, such as AST-based static analysis for the OCP [3], are often narrowly focused on a single principle. Manual case studies underscore the industrial relevance of SOLID [4], [5] but lack automation. The application of LLMs is particularly promising, given their demonstrated ability to identify related issues like code smells [6]. However, direct applications to SOLID principles remain preliminary, largely consisting of theoretical proposals rather than empirical benchmarks [7].

This reveals a critical gap between conceptual proposals and practical validation: there is no systematic, empirical benchmark of LLM performance on SOLID violation detection, evaluated across a spectrum of models, programming languages, and prompting strategies. Progress is further hampered by the absence of a standardized, public benchmark dataset designed for this specific, multi-language problem.

This paper directly addresses these gaps. We present the first systematic evaluation of four state-of-the-art LLMs—CodeLlama:70b [8], DeepSeekCoder:33b [9], Qwen2.5-Coder:32b [10], and GPT-4o-mini [11]—on their ability to detect SOLID violations across Python, Java, C#, and Kotlin. To enable this evaluation, we introduce a new benchmark dataset and test four custom prompt strategies inspired by established zero-shot, few-shot, and reasoning-based techniques. Our goal is to provide a foundational understanding of LLM capabilities and limitations in this domain.

Our key contributions are:

- 1) A systematic benchmark of LLMs for SOLID violation detection, representing the first empirical study to assess performance across four distinct models, four programming languages, and four tailored prompt strategies.
- 2) A benchmark dataset of 240 manually validated code snippets for SOLID violation detection, covering all five principles at three difficulty levels and uniquely providing both violating and refactored code versions.
- 3) An open-source replication package containing all data, prompts, evaluation scripts, and raw model outputs to ensure full reproducibility, publicly available at: <https://doi.org/10.5281/zenodo.17008546>

II. RELATED WORK

Previous works link the adherence of SOLID principles with maintainability, where violations correlate with more code smells and reduced quality. Ampatzoglou et al. [1] show SRP violations raise common code smells, while Turan and Tanrıöver [2] quantify maintainability gains (e.g., analyzability) from adhering to SOLID guidelines; and Yanakiev et al. [5] demonstrate this by improving a legacy C++ system via DIP refactoring. A complementary thread targets design-time validation. Oktafiani and Hendradjaya [12] propose class-diagram compliance metrics, while Chebanyuk and Markov [13] offer logic-based verification for class diagrams. While this thread confirms the importance of SOLID, its methods are either manual or confined to design-time artifacts, not automated, code-level detection.

A second thread focuses on automating code quality checks. Traditional static analysis tools such as SonarQube [14] and Codacy [15] detect general quality issues but are known for high false positive/negative rates [16], [17]. More recent tools like DeepCode [18] and Amazon CodeGuru [19] leverage machine learning to mitigate these issues. Even advanced approaches like Intelligent Code Analysis Agents (ICAA) that combine LLMs with static analysis [20] focus on general-purpose analysis. Thus, this thread lacks tools with a comprehensive focus on detecting violations across all five SOLID principles directly from code.

The most recent thread explores LLMs for understanding high-level design. The potential is clear, as prompt engineering techniques like few-shot, chain-of-thought, and role-based prompting have proven effective in related tasks like code smell detection [6], [21], [22]. However, direct applications of LLMs to SOLID principles remain preliminary. For instance, Martins et al. [7] proposed a GPT-4-based GitHub bot for code reviews, but their work is a theoretical proposal without quantitative evaluation. Other studies highlight SOLID’s importance for developer understanding in ML code [23] or note its underutilization in ML pipelines [24]. Crucially, these studies underscore the relevance of SOLID but stop short of providing a systematic, empirical benchmark to evaluate LLM performance on this detection task.

In contrast to prior work, our study provides the first systematic evaluation of multiple LLMs across all five SOLID principles and four programming languages using a dedicated benchmark dataset. Unlike design-time metrics or narrowly scoped tools, we evaluate detection directly on code, using accuracy and F1 scores. By analyzing the impact of prompt strategies, we offer new perspective into how LLMs internalize software design principles, positioning them as reflective tools for assessing code quality.

III. METHODOLOGY

Our methodology consists of three parts: (1) dataset creation, (2) prompt strategy design, and (3) model-based classification. Figure 1 provides an overview of this approach, where each code snippet from our constructed dataset is analyzed by

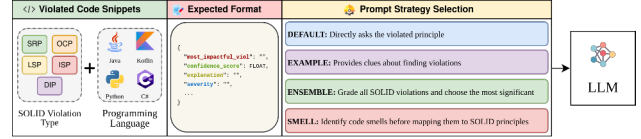


Fig. 1: Overview of the approach

an LLM via a tailored prompt to predict a potential SOLID violation.

A. Dataset Construction and Validation

A key challenge in this domain is the absence of a public benchmark for SOLID violations. To address this, we construct a new dataset covering all five principles across four languages: **Java, Python, Kotlin, and C#**.

Our creation process follows a hybrid methodology that combines LLM-based generation with manual authoring. First, we define 20 representative violation scenarios, four per principle, inspired by canonical patterns described in foundational software engineering literature [25]. For each scenario, we prompt OpenAI’s gpt-4o with structured requests specifying the violation type, target language, and complexity level. One author refined outputs for realism and clarity, implementing corresponding non-violating versions. Another author then revised the output as needed and implemented the corresponding non-violating version. A second author independently verified both versions for correctness and consistency. In case of disagreements, the first and second authors discussed and resolved conflicts through consensus.

To control for complexity, we implement each scenario at three difficulty levels: easy, moderate, and hard. We use character count and cyclomatic complexity as the proxy for difficulty. As Figure 2 shows, our assigned difficulty labels correlate strongly with both character count and cyclomatic complexity, validating our choice of proxy. The final dataset consists of 240 unique, manually validated code samples. A complete list of all 20 scenarios is available in our public replication package.

B. Prompt Engineering Strategies

We design and evaluate four prompt strategies, each requiring structured JSON output that classifies code into one of six classes (SRP, OCP, LSP, ISP, DIP, or No Violation) with a brief explanation. The baseline DEFAULT prompt is a direct zero-shot request [26]. The EXAMPLE prompt uses a few-shot approach [27], embedding one concise line per principle in the prompt to illustrate its violation (e.g., SRP: “unrelated responsibilities,” OCP: “repeated if/switch where polymorphism fits,” LSP: “subclass breaks base contract,” ISP: “fat interfaces,” DIP: “depends on concretes”). The SMELL prompt applies a two-step Chain-of-Thought process [28] where the model is first asked to identify design smells, then map them to SOLID principles. The model then scores each principle (0–5), before outputting only the most violated one. Importantly, no explicit mapping between smells and principles is given, requiring inference from the models’ prior

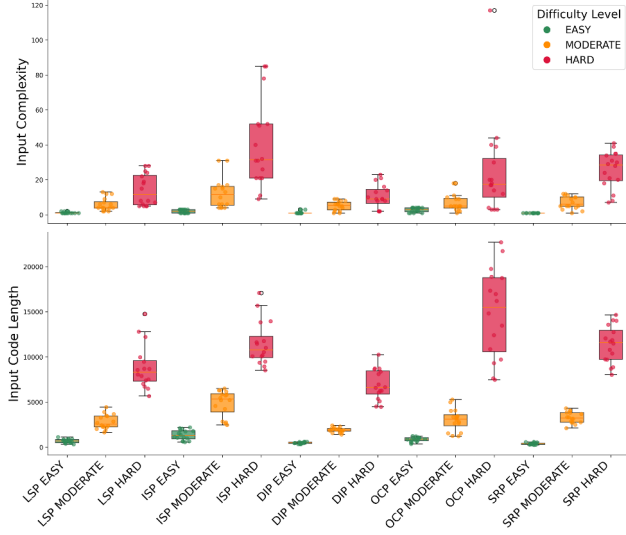


Fig. 2: Code characteristics (character count, cyclomatic complexity) by SOLID violation and difficulty.

knowledge. Finally, the ENSEMBLE strategy asks the model to score all five principles (0–5) with one-line justifications, then select and justify the single most impactful violation.¹

C. Classification Process

To evaluate model performance, we process each model output against our dataset’s ground-truth labels. We first attempt to automate this classification using tailored regular expressions designed to parse the unique output format of each prompt strategy. However, this automated approach reveals a significant challenge: LLMs frequently fail to adhere to the requested output structure.

This widespread non-adherence requires us to manually review and label 1,431 out of 3,840 total responses (37%). We perform this manual validation to resolve specific failure cases, such as when models detect multiple distinct violations, the regex fails to find a clear indicator, or language-specific response patterns emerge. This rigorous two-stage process ensures the quality and reliability of the final labels used for our analysis.²

D. Evaluation Metrics

To measure model performance, we use two standard classification metrics: Accuracy and F1-Score.

$$F1\text{-Score} = \frac{2 \times TP}{2 \times TP + FP + FN}, \quad \text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \quad (1)$$

where TP , FP , FN , and TN represent true positives, false positives, false negatives, and true negatives respectively.

¹Full prompt templates and scripts are provided in the replication package.

²The specific regular expressions and detailed labeling criteria are available in our public replication package.

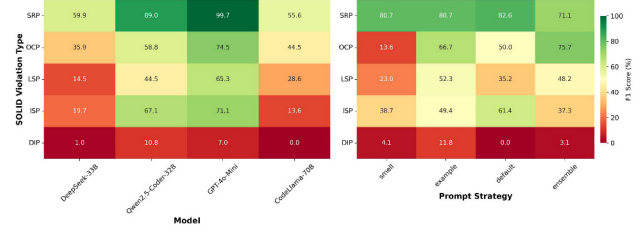


Fig. 3: F1 scores of SOLID violation detection across a) LLM models and b) prompt strategies.

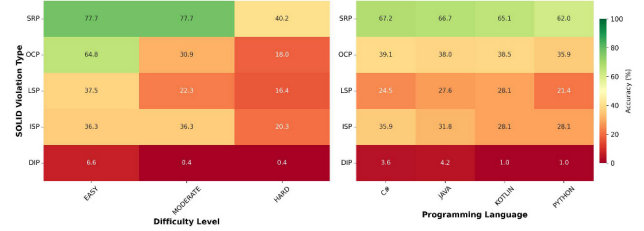


Fig. 4: Average detection accuracy for a) each sample level and b) across programming languages per SOLID violation.

E. Experiment Setup

We design our experiments to answer our research questions by systematically evaluating four LLMs [8]–[11], chosen to represent a diverse range of architectures, organizations, and sizes. All experiments are conducted at a temperature of 0 to ensure deterministic and reproducible outputs. We address each research question as follows: **RQ1** investigates detection performance by comparing the accuracy of different LLMs across all violation types. **RQ2** assesses the impact of prompt engineering strategies by measuring changes in detection success rates for each SOLID principle under varying prompt formulations. **RQ3** evaluates language-specific performance.

IV. EVALUATION

A. **RQ1:** What is the relative performance of different LLMs in detecting SOLID violations?

Our results show a stark performance hierarchy among models. GPT-4o Mini is decisively the top performer, while other models struggle significantly, especially with more complex principles. As shown in Figure 3a, GPT-4o Mini demonstrates superior performance across nearly all principles, achieving exceptional F1-scores for SRP (99.7), OCP (74.5), and ISP (71.1). Qwen2.5-Coder-32B is a distant but clear second, showing competence in detecting SRP (89.0) and OCP (58.8) but faltering on more nuanced principles like DIP (10.8). The other models perform poorly; CodeLlama-70B’s performance is weak outside of SRP (55.6), with a particularly low score for ISP (13.6), and DeepSeek-33B fails to achieve an F1-score above 40 for any principle other than SRP. For DIP, the most challenging principle, three of the four models are effectively unable to provide useful detections.

Furthermore, code complexity is a major factor. Figure 4a reveals a sharp decline in detection accuracy for all models when moving from easy to moderate and hard samples. While SRP remains relatively easy to detect regardless of complexity, accuracy on principles like DIP declines sharply, underscoring the challenge that complex code poses to current models.

B. RQ2: *How do different prompt strategies affect the ability of LLMs to detect violations of the SOLID principles?*

Prompting strategy is a critical factor, but no single strategy excels at all tasks. The ENSEMBLE and EXAMPLE strategies show strong, complementary performance, while the SMELL strategy is a consistent failure.

Figure 3b reveals a complex relationship between prompts and principles. The ENSEMBLE strategy is surprisingly effective for OCP (F1-score of 75.7), significantly outperforming all others. The EXAMPLE strategy proves most effective for LSP (52.3) and DIP (11.8), demonstrating that providing a hint is crucial for these nuanced violations. The baseline DEFAULT strategy excels at detecting SRP (82.6) and ISP (61.4), suggesting that for principles with clear structural patterns, a direct prompt is sufficient. In stark contrast, the SMELL strategy consistently underperforms, with catastrophically low scores for OCP (13.6) and LSP (23.0), suggesting that the indirect, two-step reasoning is ineffective for this task.

C. RQ3: *How does programming language affect the detection accuracy of SOLID violations?*

Detection accuracy is highly dependent on the programming language, with the structural clarity of statically-typed languages like C# and Java leading to better performance, especially for simpler principles.

Figure 4b shows that C# and Java yield the highest overall accuracy. For the most easily detected principle, SRP, they achieve scores of 67.2 and 66.7, respectively. This suggests that features common to these languages, such as explicit type declarations and formal class structures, provide clearer signals for LLMs. Kotlin follows, performing on par with Java for LSP (28.1 vs 27.6) but lagging elsewhere. Python, with its dynamic typing, consistently presents the greatest challenge, showing the lowest accuracy for four out of the five principles. This indicates that its syntactic flexibility makes design violations more ambiguous for automated tools.

D. Cross-Cutting Finding: The Impact of Code Complexity

Across all models, prompts, and languages, increasing code complexity is the greatest factor that degrades detection performance.

Figure 4a illustrates a sharp, universal decline in accuracy as samples move from EASY to MODERATE and HARD. For instance, OCP detection accuracy plummets from 64.8 on easy samples to just 18.0 on hard ones. This trend is even more pronounced for the most difficult principles; both LSP and DIP have accuracy scores below 25 for moderate and hard samples. While SRP detection remains somewhat robust, the overall pattern confirms that the selected LLMs struggle significantly to untangle design violations from general code complexity.

Why models fail: We observed three recurring failures: (i) *Principle ambiguity:* DIP and LSP require reasoning about

abstractions that are harder to infer from code snippets. As a result, models tend to over-rely on more surface-level structural cues, leading to inflated detection of SRP and ISP violations. (ii) *Two-step prompting:* The SMELL prompt requires the model to implicitly map design smells to SOLID principles without explicit guidance. This increases cognitive load and error propagation, contributing to its consistently low F1 scores. (iii) *Schema non-adherence:* Models frequently produce outputs that deviate from the expected format, necessitating manual review in 37% of cases. Additionally, we observe a sharp decline in performance as code complexity increases, suggesting that incidental complexity can obscure the design-relevant signals the models are intended to detect.

V. CONCLUSION AND FUTURE WORK

We presented the first systematic evaluation of LLMs for detecting SOLID design principle violations across four models, languages, and prompt strategies on a new, manually validated dataset. Our findings provides emerging evidence that LLM effectiveness critically depends on the model, prompt, and code context.

GPT-4o Mini emerged as the top performer, while direct, context-aware prompts (e.g., EXAMPLE) significantly outperformed abstract reasoning strategies. Statically-typed languages (C# and Java) facilitated more accurate detection than dynamically-typed languages like Python.

This work has direct implications for how we assess AI coding assistants. An LLM’s ability to reason about SOLID principles serves as a crucial proxy for its underlying “design awareness.” This proxy is vital because models lacking a grasp of these principles are likely to generate code that, while functional, degrades into less maintainable and extensible systems over time.

While this study provides insightful information, we acknowledge its limitations. Our findings are based on a synthetic dataset, which may not fully represent the complexity of industrial codebases. The results are a snapshot in time; the rapidly evolving LLM landscape may alter specific model rankings. Finally, the results may not generalize beyond the specific models, languages, and violation patterns we investigated. These limitations motivate several avenues for future work.

A key next step is to move from violation detection to automated refactoring. Future studies should task LLMs with generating corrections for the violations in our dataset. The quality of these LLM-generated solutions could then be rigorously assessed through a dual-evaluation approach: qualitatively via expert review against our manually written solutions, and quantitatively via automated test cases to verify that functional correctness is preserved. Further research should also expand this benchmark to include more models, real-world industrial code, and a wider range of design patterns.

VI. ACKNOWLEDGEMENT

We thank Rafi Çoktalaş and Muammer Buğra Kurnaz for their contributions to the initial prototype and discussions.

REFERENCES

- [1] A. Ampatzoglou, A.-A. Tsintzira, E.-M. Arvanitou, A. Chatzigeorgiou, I. Stamelos, A. Moga, R. Heb, O. Matei, N. Tsiridis, and D. Kehagias, "Applying the single responsibility principle in industry: modularity benefits and trade-offs," in *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*, 2019, pp. 347–352.
- [2] O. Turan and Ö. Ö. Tanrıöver, "An experimental evaluation of the effect of solid principles to microsoft vs code metrics," *AJIT-e: Academic Journal of Information Technology*, vol. 9, no. 34, pp. 7–24, 2018.
- [3] G. Roy, M. M. and B. A. Jacob, "Automated Verification of Open/Closed Principle: A Code Analysis Approach," in *2024 5th International Conference for Emerging Technology (INCET)*, May 2024, pp. 1–7.
- [4] A. V. Girjoaba and A. Capiluppi, "Refactoring Legacy Code Using Cleaning Up Cycles: An Experience Report," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2024, pp. 753–764.
- [5] I. Yanakiev, B.-M. Lazar, and A. Capiluppi, "Applying solid principles for the refactoring of legacy code: An experience report," *Journal of Systems and Software*, vol. 220, p. 112254, 2025.
- [6] A. R. Sadik and S. Govind, "Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3," *arXiv preprint arXiv:2504.16027*, 2025.
- [7] G. F. Martins, E. C. M. Firmino, and V. P. De Mello, "The Use of Large Language Model in Code Review Automation: An Examination of Enforcing SOLID Principles," in *Artificial Intelligence in HCI*, H. Degen and S. Ntoa, Eds. Cham: Springer Nature Switzerland, 2024, pp. 86–97.
- [8] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: <https://arxiv.org/abs/2308.12950>
- [9] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: <https://arxiv.org/abs/2401.14196>
- [10] Qwen, :, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 technical report," 2025. [Online]. Available: <https://arxiv.org/abs/2412.15115>
- [11] OpenAI, "Gpt-4o: Openai's new omni model," <https://openai.com/index/gpt-4o>, 2024, accessed: 2025-05-16.
- [12] I. Oktafiani and B. Hendradjaya, "Software Metrics Proposal for Conformity Checking of Class Diagram to SOLID Design Principles," in *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, Nov. 2018, pp. 1–6.
- [13] E. Chebanyuk and K. Markov, "An approach to class diagrams verification according to SOLID design principles," in *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Feb. 2016, pp. 435–441.
- [14] S. SA, "Sonarqube - continuous code quality," <https://www.sonarsource.com/products/sonarqube/>, 2024, accessed: 2025-05-16.
- [15] I. Codacy, "Codacy - automated code review and quality monitoring," <https://www.codacy.com/>, 2024, accessed: 2025-05-16.
- [16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
- [17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.
- [18] S. Ltd., "Snyk code (formerly deepcode)," <https://snyk.io/product/snyk-code/>, 2024, accessed: 2025-05-16.
- [19] I. Amazon Web Services, "Amazon codeguru reviewer," <https://aws.amazon.com/codeguru/>, 2024, accessed: 2025-05-16.
- [20] G. Fan, X. Xie, X. Zheng, Y. Liang, and P. Di, "Static code analysis in the ai era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents," *arXiv preprint arXiv:2310.08837*, 2023.
- [21] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "ismell: Assembling llms with expert toolsets for code smell detection and refactoring," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1345–1357.
- [22] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," in *Generative ai for effective software development*. Springer, 2024, pp. 71–108.
- [23] R. Cabral, M. Kalinowski, M. T. Baldassarre, H. Villamizar, T. Escovedo, and H. Lopes, "Investigating the Impact of SOLID Design Principles on Machine Learning Code Understanding," in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, ser. CAIN '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 7–17.
- [24] L. López, C. Gómez, and C. Ayala, "Insights on the Use of Software Design Principles in Machine Learning Pipelines," in *Product-Focused Software Process Improvement*, D. Pfahl, J. Gonzalez Huerta, J. Klünder, and H. Anwar, Eds. Cham: Springer Nature Switzerland, 2025, pp. 139–155.
- [25] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.
- [26] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.
- [27] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [28] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.