Prompting Matters: Assessing the Effect of Prompting Techniques on LLM-Generated Class Code

Adam Yuen, John Pangas, Md Mainul Hasan Polash, Ahmad Abdellatif
University of Calgary
Calgary, Canada
{adam.yuen, john.pangas, mdmainulhasan.polash, ahmad.abdellatif}@ucalgary.ca

I. Abstract

The field of software engineering and coding has undergone a significant transformation. The integration of large language models (LLMs), such as ChatGPT, into software development workflows is changing how developers at all skill levels approach coding tasks. Leveraging the capabilities of LLMs, developers can now implement functionalities, fix bugs, and address reviewers' comments more efficiently. However, prior research shows that the effectiveness of LLM-generated code is heavily influenced by the prompting strategy used. Furthermore, generating code at the class level is significantly more complex than at the method level, as it requires maintaining consistency across multiple methods and managing class state. Therefore, this study evaluates the impact of four prompting strategies (i.e., Zero-Shot, Few-Shot, Chain-of-Thought, and Chain-of-Thought-Few-Shot) on GPT and Llama3 in generating class-level code. It assesses the functional correctness and the quality characteristics of the generated code. To better understand how errors differ by prompting strategy, a qualitative analysis of the generated code is conducted for test cases that fail. The findings show that strategies incorporating more contextual guidance (Few-Shot, Chain-of-Thought, and Chain-of-Thought Few-Shot) outperform Zero-Shot prompting by up to 25% in functional correctness, 31% in BLEU-3 score, and 50% in ROUGE-L, while also producing code that is more readable and maintainable. The results also indicate that procedural logic and control flow errors are the most prominent, accounting for 31% of all errors. This study provides valuable insights to guide future research in developing techniques and tools that enhance the quality and reliability of LLM-generated code for complex software development tasks.

II. Introduction

Large Language Models (LLMs) are rapidly transforming the software development landscape, where tools such as OpenAI's ChatGPT, GitHub Copilot, and Claude AI are often leveraged by developers to ease their coding workflows [1], [2]. It has become increasingly common for these tools to be used to generate code as their ability to reason continues to increase alongside their popularity. As these models are used increasingly by software engineers, questions surrounding the quality, correctness, and reliability of LLM-generated code have grown in importance. Recent work demonstrates that the performance of LLMs, particularly in code generation, is heavily dependent on prompting [3]. Moreover, recent work in this space has primarily focuses on the functional correctness of generated code, often using function-level prompts to evaluate whether LLMs can accurately implement isolated methods [4]–[7]. However, generating class-level code is more complex, as it requires the model to reason about inter-method dependencies, maintain a coherent state through instance variables, and correctly implement object-oriented design principles such as encapsulation and inheritance [8].

Therefore, this paper presents the first attempt to evaluate the impact of prompting strategies on LLM-generated code at the class-level, as well as which prompting strategy is optimal for code generation. To do so, we leverage the ClassEval class-level code generation benchmark [9] to assess the impact of four prompting strategies (Zero-Shot, Few-Shot, Chain-of-Thought, and Chain-of-Thought-Few-Shot) on GPT-4o and Llama3-70B in class-level code generation. Our work investigates the following research questions (RQs):

RQ1: How do different prompt strategies impact GPT's performance in generating class-level code? Our study finds that Few-Shot, Chain-of-Thought, Chain-of-Thought-Few-Shot prompt strategies perform the best, achieving 82.1%, 84.8%, and 83.7% pass rates, respectively.

RQ2: How do different prompt strategies affect the quality characteristics of LLM-generated code? The results demonstrate that these prompts generate higher-quality code, exhibiting improved complexity, nesting, and documentation compared to an equivalent, human-authored baseline while still maintaining similarity, supported by NLP metrics. Our results also demonstrate that LLM-generated code tends to be less verbose, with 5-10 fewer median number of code lines when compared to a human-generated baseline.

RQ3: What types of errors occur in LLM-generated code under different prompt strategies? From the analysis of the errors in RQ1, it is observed that Chain-of-Thought-based prompts perform the best in terms of reducing several error types, such as structure and logic based failures, further supporting that additional context improves the generated code output. For instance, across all categories of failures, we observe Zero-Shot prompting results in 3 times the overall errors versus other prompt strategies, at 536 total failures versus ~180 in all other strategies.

III. Experiment Design

The main goal of our study is to assess the impact of different prompting strategies on LLM-generated code. In this section, we describe the dataset and the process of constructing the prompting strategies.

A. Dataset

To the best of our knowledge, there are no existing datasets specifically designed with prompts for generating class-level code. To construct a dataset containing such prompts along with their corresponding code, we leverage ClassEval dataset proposed by Yu et al. [9] that has been used by prior SE work [10], [11]. ClassEval contains 100 classes that present a variety of object-oriented programming challenges, such as a Blackjack game and a utility for determining the longest word in a sentence. ClassEval is a manually curated benchmark developed to evaluate LLMs on more complex class-level Python coding tasks; unlike prior datasets, which primarily focus on method-level problems. Furthermore, ClassEval provides test cases for each instance, enabling robust evaluation of functional correctness by comparing LLM-generated implementations to the expected outputs. These features make it well-suited for our study. In the following section, we describe the prompting strategies used in this study and outline the steps taken to construct the prompts.

B. Prompt Strategies

We study four prompting strategies: Zero-Shot, Few-Shot, Chain-of-Thought, and Chain-of-Thought-Few-Shot. These strategies have been used in a variety of SE work [12]–[16].

- **Zero-shot prompting** provides only a task instruction without examples, relying on the model's pre-trained knowledge. Our Zero-Shot prompt includes a class-level docstring and a structural skeleton outlining method names and headers. We adapt ClassEval's *class skeletons* [12], [14] by removing parameter descriptions, example inputs/outputs, and implementation hints to adhere to zero-shot format.
- **Few-shot prompting** builds off of zero shot by adding example input and output to guide the model. We reincorporate removed elements from the class skeleton (e.g., parameter descriptions, example I/O) and append a separate ClassEval prompt and its canonical solution [9]. This enriches context and provides the LLM with a reference solution to a similar task.
- **Chain-of-Thought (CoT) prompting** introduces intermediate reasoning steps to guide step-by-step generation [17]. To construct CoT prompts for all classes, two authors independently write prompts for each of the 100 coding problems in ClassEval. Next, the annotators validate all of the prompts together, scrutinizing each one and merging them. In case of disagreement, a third author engaged in the discussion to reach an agreement. Each prompt begins with a task overview and is followed by instructions breaking down how to implement each method.
- **Chain-of-Thought Few-shot (CoT Few-Shot) prompting** combines the benefits of CoT and Few-Shot prompting [14]. Each prompt includes a CoT-style explanation for the target task, along with a full CoT prompt and solution for a different ClassEval problem. This strategy aims to improve reasoning and structure by offering an in-context worked example. As one prompt is reserved as the example, Few-Shot and CoT Few-Shot strategies run on 99 tasks instead of 100.

Our study evaluates 396 prompts, with 100 of both Zero-Shot and CoT, and 99 of both Few-Shot and CoT Few-Shot. We remove the held out class used as an example in CoT Few-Shot and Few-Shot from both Zero-Shot and CoT from Zero-Shot and CoT. Thus, the final dataset contains 396 prompts (99 prompts for each strategy), which are available at [18]. After constructing the prompts, we evaluate the code generated for each one by utilizing a Python pipeline that automates the process of gathering prompts and submitting them to the LLMs. Specifically, the pipeline iterates through each prompt in our dataset and sends it to the LLMs sequentially. GPT-4o and Llama3 are configured with a temperature of 0.5 to strike a balance between deterministic behavior and what a typical user would see, a token limit of 1000 (2000 for Llama3 due to API differences), and a single completion per prompt. The outputs are collected in a large CSV file for analysis to answer our RQs.

IV. Results

In this section, we present the results of our study. We describe our motivation, methodology to answer the questions, and results for each research question.

A. RQ1: How do different prompt strategies impact GPT's performance in generating class-level code?

Motivation: Understanding the correctness of LLM-generated code is fundamental to evaluating the reliability and utility of LLMs in programming contexts. This RQ examines code correctness by assessing how accurately the generated code aligns with functional requirements. The results will guide practitioners in selecting the appropriate prompt for class implementation.

Approach: To determine the correctness of the LLM-generated code, our study leverages the test suite from ClassEval. We reiterate that each coding problem outlined in the ClassEval dataset contains a series of test cases; every LLM-generated code snippet is tested against its appropriate test class. We run the corresponding test cases against the LLM-generated class. We report the mean pass rate across all test cases for all classes for each prompting strategy.

Results: Table I presents GPT's performance for each prompting strategy in terms of test pass rate. Overall, CoT achieved the highest functional correctness with an average pass rate of 84.%5 to 84.8%, followed closely by CoT Few-Shot at

TABLE I: Pass Rates and Correct Generated Class Counts by Prompting Strategy for GPT and Llama3.

Strategy	Pass Rate (%)		Correct Classes (out of 99)	
	GPT-4o	Llama3	GPT-4o	Llama3
Zero-Shot	57.8	59.8	21	19
Few-Shot	82.1	78.8	38	31
CoT	84.8	84.5	38	37
CoT Few-Shot	83.7	81.9	40	38

TABLE II: Results for NLP Metrics (BLEU-3, ROUGE-L) by Prompting Strategy and Model

Strategy	BLEU-3 (%)		ROUGE-L (%)	
	GPT	Llama3	GPT	Llama3
Zero-Shot	8.9	9.6	32.7	45.4
Few-Shot	10.3	8.3	31.0	43.9
CoT	37.9	26.0	73.1	61.3
CoT Few-Shot	39.3	29.0	72.6	65.9

81.9% - 83.7% and Few-Shot at 82.1%. These results suggest that structured prompting strategies that encourage step-by-step reasoning, whether with or without examples, can significantly enhance the LLM's ability to generate code that passes behavioral tests. On the other hand, Khojah et al. [7] report that using a few-shot prompting strategy achieved the best results for class-level code generation with GPT-4o on their dataset. This difference might be due to the fact that generating class-level code is more complex than generating method-level code.

Zero-Shot prompting resulted in a substantially lower pass rate of 57.8%- 59.8%, as shown in Table I. Compared to Zero-Shot, the three prompting strategies with pass rates in the 80% range incorporate additional information such as intermediate reasoning steps and/or example-driven context, indicating a limited ability of LLMs to generate functionally correct code without contextual guidance or demonstration. These trends support the hypothesis that more informative prompts lead to better generalization and correctness in LLM-generated code.

B. RQ2: How do different prompt strategies affect the quality characteristics of LLM-generated code?

Motivation: While functional correctness is essential for code reliability, software engineers must also consider code quality characteristics such as complexity and maintainability. These characteristics affect how easily code can be reviewed, debugged, and extended, all critical factors in software development. Thus, beyond evaluating whether generated code works, it is crucial to investigate its alignment with best practices in software engineering. In this RQ, we explore the LLM-generated code characteristics for each prompting strategies.

Approach: To assess the LLM-generated code's quality, we leverage SciTools' Understand tool to analyze both the functionally correct and incorrect code snippets generated during RQ1. Specifically, we compute size in terms of lines of code, complexity in terms of the maximum cyclomatic complexity for a class, and readability based on the number of comments. To put our results into perspective, we compute the code quality characteristics of the canonical solution set (oracle) from ClassEval. Furthermore, we compute BLEU3 score and ROUGE-L F1 score [19], [20] of the generated code with the expected solution to quantify how closely the LLM-generated code resembles the oracle.

Results: Table III presents the median values of code characteristics for correct and incorrect LLM-generated code. Overall, the LLM-generated code, regardless of its functional correctness, exhibits complexity levels comparable to human-generated code, with a median complexity of 3 observed in both correct code snippets and the oracle We also observe that functionally correct code snippets contain fewer lines of code and fewer comments compared to both incorrect LLM-generated code and the oracle. We report that the oracle contained a median of 5 lines of comments per class, a mere 14% of what Few-Shot can generate. Additionally, on average, correct code contains 25% fewer lines and 12% fewer comments than incorrect LLM-generated code, with the oracle reporting a median of 33 code lines. Interestingly, both the CoT and CoT Few-Shot prompting strategies result in zero code comments, regardless of whether the generated code is correct or incorrect. One potential reason behind this observation is that these prompting strategies focus heavily on guiding the model's reasoning process through natural language rather than encouraging documentation within the code itself, which may inadvertently deprioritize in-line commenting during generation. On the other hand, LLMs tend to include more code comments when using Zero-Shot and Few-Shot prompting strategies, likely in an effort to clarify the code as it is being generated.

Table II presents the similarity between LLM-generated code and the oracle (human-written code) for each prompt strategy. From the table, we observe that the code generated using the CoT and CoT Few-Shot strategies is most similar to human-generated code in terms of BLEU-3 and ROUGE-L scores, compared to Zero-Shot and Few-Shot strategies. This might be due to the fact that CoT prompting encourages the model to reason through the problem step-by-step, leading to more structured and coherent code generation that better aligns with human coding practices.

Based on our results, we recommend that practitioners use the CoT and CoT Few-Shot strategies for generating class-level code when the goal is to produce code that closely resembles existing project code, as these strategies yield higher BLEU-3 and ROUGE-L scores. However, if their objective is to obtain alternative implementations with more extensive code documentation, the Few-Shot strategy may be more suitable.

C. RQ3: What types of errors occur in LLM-generated code under different prompt strategies?

Motivation: After understanding the quantitative metrics for the LLM-generated code, we want to analyze the nature of the code's errors. Thus, in this RQ, we analyze the types of errors and how they vary by prompting strategy. This analysis offers insights into the specific challenges LLMs face in code generation and helps identify which prompting strategies are more prone to particular types of mistakes, ultimately guiding the design of more effective prompts.

TABLE III: Comparison of Code Quality Metrics by Prompting Strategy and Model

Category	GPT-4o						Llama3					
	Correct Code			Incorrect Code			Correct Code			Incorrect Code		
	Cyclo.	Code	Comm.	Cyclo.	Code	Comm.	Cyclo.	Code	Comm.	Cyclo.	Code	Comm.
Zero-Shot	3	20.5	22	3	30	27	3	25	18	3	32	15
Few-Shot	3	23	33.5	3	30	35	3	26	29	3	32.5	38
CoT	2	25.5	0	3	31	0	3.5	28	0	3	28	0
CoT + Few-Shot	3	23.5	0	3	31	0	3	27	0	3	36	0

Approach: To conduct our qualitative analysis, we examine the test outputs produced in RQ1 to classify the functional failures in LLM-generated code. We randomly sampled failed test cases from the GPT-4o model with a 95% confidence level and a 5% confidence interval from each prompting strategy, resulting in a sample size of 1095. One author performed open coding by manually inspecting the test outputs and the corresponding LLM-generated code. The root cause of each failure was identified and categorized based on distinct types of recurring errors observed in the generated code.

Results: Table IV presents 16 error types, grouped into six main categories. The table also shows the distribution of each error across the four prompting strategies. Our study reveals a more specific set of error categories related to class-level code generation, such as Missing Methods. Below, we describe the six categories along with their associated error types.

Type and Structure Errors: This category encompasses errors such as *Incorrect Data Types*, where generated code deviates from the expected formats (e.g., producing an integer instead of a string); *Improper Use of Collections*, where unsuitable data structures are used; and *Mismatched Return Types*, where return types are unexpected - for instance, returning None instead of a boolean. Type and Structure errors were most prevalent under Zero-Shot prompting (84 instances) and were notably mitigated in other strategies.

Logic and Control Flow Errors: This category includes *Incorrect Procedural Logic* errors which occurred due to flawed implementation steps, such as dividing by n instead of $n-1$; *Faulty Conditional Branching* errors, which involved incomplete or invalid condition expressions, and *Inadequate Loop Control* errors where loop termination conditions were missing or incorrect, resulting in infinite or prematurely terminated loops. Logic and Control Flow Errors were prevalent across all prompting strategies, with the majority stemming from *Incorrect Procedural Logic*, accounting for 14% to 34% of total errors per strategy.

Method and Attribute Errors: Outlines errors due to *Missing Methods*, *Incorrect Method Signatures*, where functions had wrong parameter configurations; and *Incorrect Attribute Naming*, which involved misnaming variables or object properties, such as using conn instead of the correct identifier connection. Zero-Shot prompting showed a particularly high incidence of *Incorrect Attribute Naming* errors (93 instances), far exceeding those seen in other prompting approaches.

Input Handling Errors: This category included *Parsing Errors*, such as failures in processing date or string formats, *Insufficient Input Validation*, where checks for null or invalid inputs were absent, and *Incorrect Input Assumptions*, where the model assumed the presence of inputs (such as specific dictionary keys) without verification. These errors were especially common in Zero-Shot (74 instances) and were significantly reduced with few-shot prompting.

Output Formatting and Content Errors: Errors in this category include *Improper Output Formatting*, such as returning a tuple instead of a string, and *Incorrect Output Content*, where output structures are valid but are semantically flawed or incomplete. These errors persisted across all strategies, with most recording 40 or more instances each.

Dependency and External Library Errors: This involved *Missing Imports*, where required external libraries were used but not declared, and *Incorrect Library Usage*, such as using `tdatetime.now()` instead of `datetime.datetime.now()`. Zero-Shot prompting accounted for a disproportionate number of these errors (72 instances), while other strategies showed significantly fewer occurrences (5 or fewer).

Following this, the authors analyzed 60% of the failed test case classes from the Llama3 model to assess how well our taxonomy of failure types generalizes beyond the GPT-generated code. We observed similar patterns in the reasons for failure, and therefore report detailed findings for only 30 of Llama3's classes. A full analysis of all Llama3-generated classes is available in our replication package.

Overall, the findings support that prompting strategies that incorporate reasoning or illustrative examples substantially improve LLM performance by reducing structural, logical, and integration-related errors.

V. Threats to Validity

To construct the prompts for our study, two authors manually examined the 100 coding problems from ClassEval, which might involves subjective design decisions that could influence the outcome. To alleviate this threat, the annotators independently created the prompts and then jointly reviewed and resolved discrepancies. In cases of disagreement, the prompt was discussed with a third author to reach an agreement.

We use GPT-4o and Llama3 70B to assess the impact of different prompting strategies on the LLM-generated code. Furthermore, we use the ClassEval dataset to conduct our study. Thus, our results and observations might not be generalized to other facets of software engineering. Thus, our results may not generalize to other datasets and LLMs with different architectures, training data, or capabilities. We plan

TABLE IV: Frequency of Generated Code Failures by Error Type and Prompting Strategy

Main Category	Error Type	Zero-Shot	Few-Shot	Chain-of-Thought	Chain-of-Thought-Few-Shot
Type and Structure Errors	Incorrect Data Types	24	0	0	10
	Improper Use of Collections	44	9	0	10
	Mismatched Return Types	16	10	11	3
Logic and Control Flow Errors	Incorrect Procedural Logic	77	**67**	**66**	**67**
	Faulty Conditional Branching	23	10	14	14
	Inadequate Loop Control	1	1	3	0
Method and Attribute Errors	Incorrect Attribute Naming	**93**	0	2	8
	Missing Methods	3	1	6	1
	Incorrect Method Signatures	19	18	14	0
Input Handling Errors	Parsing Errors	0	1	2	1
	Insufficient Input Validation	7	1	6	9
	Incorrect Input Assumptions	67	8	15	12
Output Formatting and Content Errors	Improper Output Formatting	33	10	14	10
	Incorrect Output Content	48	32	31	32
Dependency and External Library Errors	Missing Imports	72	5	5	4
	Incorrect Library Usage	9	10	0	6
Total Failures		**536**	183	189	187

(and encourage others) in the future to replicate our study using a wider variety of datasets and LLMs.

VI. Related Work

As LLMs continue to advance, they are being applied to an increasingly wide range of software engineering tasks, including code translation [21], code review [22], and unit test generation [3]. Specifically, code generation has emerged as a growing area of research, with interest in both how to evaluate model outputs and how to guide code generation effectively [6], [23]–[25]. Building on this, prior work has examined both fine-tuning and prompt engineering strategies. For example, Wang et al. [26] compared prompt tuning and fine tuning using CodeBERT and CodeT5 across tasks such as defect prediction, summarization, and code translation. Similarly, Shin et al. [27] compared GPT-4 with both prompt-engineered and fine-tuned models, concluding that fine-tuned models generally perform better and emphasize the continued need for human involvement in optimizing prompts.

Other studies have shown that LLMs are sensitive to prompt variations during code generation [28]. Gao et al. [29] examined factors affecting in-context learning, such as example order, selection, and quantity, and found that the similarity and diversity of examples significantly influence code understanding and generation by LLMs. The work closest to ours is by Khojah et al. [7], which examines function-level prompts from CoderEval, focusing on similarity, complexity, and code smells. Du et al. [30] explores holistic, incremental, and compositional prompting strategies, finding that most models perform better when generating classes method by method.

To the best of our knowledge, no prior work has evaluated the impact of four prompting strategies on the performance of LLMs at the class-level code granularity. Furthermore, our evaluation assesses the quality characteristics of LLM-generated code for each prompting strategy and analyzes the corresponding errors in the generated output. Overall, our work complements existing studies on code generation by providing a thorough evaluation of different prompting strategies and their effects on the quality and characteristics of class-level code generated by LLMs.

VII. Conclusion

In this paper, we present a systematic evaluation of Zero-Shot, Few-Shot, CoT, and CoT Few-Shot for class-level code generation using GPT-4o and Llama3 70B. We leverage the ClassEval benchmark to develop prompts that reflect real-world SE tasks, and to evaluate the LLM-generated code. Specifically, we investigate how different prompting strategies affect functional correctness, code quality characteristics, and the types of errors produced by these strategies. Our results show that CoT and CoT Few-Shot consistently achieved the highest correctness, with pass rates exceeding 81%, while Zero-Shot lagged significantly at approximately 58% for both Llama3 and GPT-4o. Additionally, CoT and CoT Few-Shot strategies also produced code with the highest similarity to human-written code, achieving BLEU-3 scores up to 39.3% and ROUGE-L scores over 61.3%, while maintaining comparable complexity and structure. Furthermore, based on analysis of all errors, Zero-Shot prompting accounted for 49% of all the errors among all strategies. Specifically, Logic and Control Flow errors emerged as the most frequent error type for 31% of the total errors from all strategies.

Ultimately, our study provides insights into how prompt design influences model performance on class-level coding tasks. These results demonstrate the importance of prompt design in maximizing the capabilities of large language models for code generation and particularly the addition of context into prompts. In the future, we plan to expand our study by including more LLMs and diversifying our dataset to include a wider variety of inputs and problems, as well as different programming languages. Furthermore, we plan to explore how the different prompting strategies extend to tasks like code modification and refactoring, as well as assessing how prompting for well-documented code impacts both correctness and maintainability.

References

[1] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, “The impact of ai on developer productivity: Evidence from github copilot,” 2023. [Online]. Available: https://arxiv.org/abs/2302.06590

[2] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[3] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, “On the evaluation of large language models in unit test generation,” 2024. [Online]. Available: https://arxiv.org/abs/2406.18181

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[5] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, “Program synthesis with large language models,” 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[6] Y. Zhang, W. Zhang, D. Ran, Q. Zhu, C. Dou, D. Hao, T. Xie, and L. Zhang, “Learning-based widget matching for migrating gui test cases,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. ACM, Feb. 2024, p. 1–13. [Online]. Available: http://dx.doi.org/10.1145/3597503.3623322

[7] R. Khojah, F. G. de Oliveira Neto, M. Mohamad, and P. Leitner, “The impact of prompt programming on function-level code generation,” 2024. [Online]. Available: https://arxiv.org/abs/2412.20545

[8] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, “Lost in translation: A study of bugs introduced by large language models while translating code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. ACM, Apr. 2024, p. 1–13. [Online]. Available: http://dx.doi.org/10.1145/3597503.3639226

[9] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation,” 2023. [Online]. Available: https://arxiv.org/abs/2308.01861

[10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, Dec. 2024. [Online]. Available: https://doi.org/10.1145/3695988

[11] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, “A survey of large language models for code: Evolution, benchmarking, and future trends,” 2024. [Online]. Available: https://arxiv.org/abs/2311.10372

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[13] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[14] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” 2023. [Online]. Available: https://arxiv.org/abs/2205.11916

[15] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, “Least-to-most prompting enables complex reasoning in large language models,” 2023. [Online]. Available: https://arxiv.org/abs/2205.10625

[16] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, “Scaling instruction-finetuned language models,” 2022. [Online]. Available: https://arxiv.org/abs/2210.11416

[17] A. K. Lampinen, I. Dasgupta, S. C. Y. Chan, K. Matthewson, M. H. Tessler, A. Creswell, J. L. McClelland, J. X. Wang, and F. Hill, “Can language models learn from explanations in context?” 2022. [Online]. Available: https://arxiv.org/abs/2204.02329

[18] Anonymus, “Github - senoryuen/prompt-quality-study,” 07 2025. [Online]. Available: https://github.com/SenorYuen/Prompt-Quality-Study

[19] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL ’02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[20] C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/

[21] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, “Lost in translation: A study of bugs introduced by large language models while translating code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. ACM, Apr. 2024, p. 1–13. [Online]. Available: http://dx.doi.org/10.1145/3597503.3639226

[22] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, “Exploring the potential of chatgpt in automated code refinement: An empirical study,” 2023. [Online]. Available: https://arxiv.org/abs/2309.08221

[23] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” 2023. [Online]. Available: https://arxiv.org/abs/2305.01210

[24] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, “Gptsniffer: A codebert-based classifier to detect source code written by chatgpt,” *Journal of Systems and Software*, vol. 214, p. 112059, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224001043

[25] P. Bareiß, B. Souza, M. d’Amorim, and M. Pradel, “Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code,” Jun. 2022, arXiv:2206.01335 [cs]. [Online]. Available: http://arxiv.org/abs/2206.01335

[26] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, “Prompt tuning in code intelligence: An experimental evaluation,” *IEEE Transactions on Software Engineering*, 2023.

[27] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, “Prompt engineering or fine-tuning: An empirical assessment of llms for code,” 2025. [Online]. Available: https://arxiv.org/abs/2310.10508

[28] X. Gao, A. Sinha, and K. Das, “Learning to search effective example sequences for in-context learning,” 2025. [Online]. Available: https://arxiv.org/abs/2503.08030

[29] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, “What makes good in-context demonstrations for code intelligence tasks with llms?” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Sep. 2023, p. 761–773. [Online]. Available: http://dx.doi.org/10.1109/ASE56229.2023.00109

[30] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “Evaluating large language models in class-level code generation,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639219