Published: 19 June 2025
Accepted: 14 January 2025
Received: 13 September 2024

RESEARCH-ARTICLE

Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models

YANLIN WANG, Sun Yat-Sen University, Guangzhou, Guangdong, China

TIANYUE JIANG, Sun Yat-Sen University, Guangzhou, Guangdong, China

MINGWEI LIU, Sun Yat-Sen University, Guangzhou, Guangdong, China

JIACHI CHEN, Sun Yat-Sen University, Guangzhou, Guangdong, China

MINGZHI MAO, Sun Yat-Sen University, Guangzhou, Guangdong, China

XILIN LIU, Huawei Technologies Co., Ltd., Shenzhen, Guangdong, China

View all

Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models

YANLIN WANG, Sun Yat-sen University, China
TIANYUE JIANG, Sun Yat-sen University, China
MINGWEI LIU*, Sun Yat-sen University, China
JIACHI CHEN, Sun Yat-sen University, China
MINGZHI MAO, Sun Yat-sen University, China
XILIN LIU, Huawei Cloud Computing Technologies, China
YUCHI MA, Huawei Cloud Computing Technologies, China
ZIBIN ZHENG, Sun Yat-sen University, China

Large language models (LLMs) have brought a paradigm shift to the field of code generation, offering the potential to enhance the software development process. However, previous research mainly focuses on the accuracy of code generation, while coding style differences between LLMs and human developers remain under-explored. In this paper, we empirically analyze the differences in coding style between the code generated by mainstream LLMs and the code written by human developers, and summarize coding style inconsistency taxonomy. Specifically, we first summarize the types of coding style inconsistencies by manually analyzing a large number of generation results. We then compare the code generated by LLMs with the code written by human programmers in terms of readability, conciseness, and robustness. The results reveal that LLMs and developers exhibit differences in coding style. Additionally, we study the possible causes of these inconsistencies and provide some solutions to alleviate the problem.

CCS Concepts: • **Software and its engineering** → *Software evolution*; **Automatic programming**.

Additional Key Words and Phrases: Code generation, Coding style inconsistency, Large language models

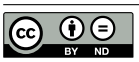
ACM Reference Format:
Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE032 (July 2025), 23 pages. https://doi.org/10.1145/3715749

1 Introduction

Code generation is to automatically generate code snippets that align with given requirements, which plays a vital role in the software engineering domain [6, 8, 15, 23, 26–28, 31–33, 38, 40, 50, 54, 62, 78, 85, 86, 89, 90, 93]. Recently, the advent of large language models [18, 39, 52], such as CodeLlama [60], StarCoder [39], Codex [56] and GPT-4 [2], has greatly advanced the performance of code generation. These models have demonstrated remarkable capabilities in code

*Corresponding Author

Authors' Contact Information: Yanlin Wang, Sun Yat-sen University, Zhuhai, China, wangylin36@mail.sysu.edu.cn; Tianyue Jiang, Sun Yat-sen University, Zhuhai, China, jiangty9@mail2.sysu.edu.cn; Mingwei Liu, Sun Yat-sen University, Zhuhai, China, liumw26@mail.sysu.edu.cn; Jiachi Chen, Sun Yat-sen University, Zhuhai, China, liumw26@mail.sysu.edu.cn; Mingzhi Mao, Sun Yat-sen University, Zhuhai, China, mcsmmz@mail.sysu.edu.cn; Xilin Liu, Huawei Cloud Computing Technologies, Shenzhen, China, liuxilin3@huawei.com; Yuchi Ma, Huawei Cloud Computing Technologies, Shenzhen, China, mayuchi1@huawei.com; Zibin Zheng, Sun Yat-sen University, Zhuhai, China, zhzibin@mail.sysu.edu.cn.



ACM 2994-970X/2025/7-ARTFSE032
https://doi.org/10.1145/3715749

generation, significantly enhancing software development efficiency [94]. However, while previous studies primarily focus on enhancing the accuracy of LLM-based code generation, an equally important aspect—coding style—remains under-explored. Understanding the differences in coding style between LLMs and human developers is essential, as it directly impacts code readability, maintainability, and overall software quality [49].

There are several previous works related to coding style [9, 48, 49, 55, 57]. Oman et al. [55] proposed a programming style taxonomy, including typographic style, control structure style, and information structure style. However, some of these guidelines are outdated, such as the use of Goto statements, which are now rarely used in modern programming. To improve code formatting style, tools like CODEBUFF [57], an automatic code formatter, and STYLE-ANALYZER [48], which fixes formatting inconsistencies, have been developed. Mi et al. [49] expanded the scope by using hierarchical agglomerative clustering to measure stylistic inconsistency, considering not only formatting but also stylistic metrics related to code readability and features specific to the C/C++ programming languages. More recently, DUETCS [9] was proposed for coding style transfer. This work considers a broader range of coding style features, categorizing them into text style (formatting and naming conventions) and structure style (code blocks ordering and preferences of control flow statements). These works provide a preliminary foundation and inspiration for examining coding styles. **In our work, we adopt the definition of code style in previous research [7, 49, 55, 80]: Code style can be understood as a personal habit in writing source code, reflecting individual preferences in aspects like physical layout, algorithms, etc.** However, there remain several gaps. **(i)** Existing work offers a generalized definition of code style but lacks detailed and comprehensive classification. **(ii)** No existing studies have yet analyzed the differences in coding style between mainstreams LLMs and human developers. **(iii)** A comparative analysis of coding styles across different LLMs is still missing.

In this paper, our aim is to fill these gaps. ① Firstly, we conduct extensive manual analysis to categorize various types of coding style inconsistencies. We compare the code generation results of five mainstream LLMs, including four Code LLMs and a state-of-the-art general-purpose LLM (GPT-4 [2]), against the ground truths of CoderEval [81] benchmark. We annotate the results and perform open coding to develop a comprehensive taxonomy of coding style inconsistencies, achieving a detailed classification of code style, which addresses **(i)** and **(ii)**. [1] ② Secondly, we analyze the distribution of inconsistencies, including ratios, frequencies, and differences across the five LLMs, addressing **(iii)**. ③ Thirdly, we compare the generated code with human-written code based on readability, conciseness, and robustness. ④ Finally, we experiment on several prompting strategies to explore methods to improve the coding style of LLMs.

Through extensive experiments and evaluation, we have obtained the following results: ❶ We propose the first taxonomy of coding style inconsistencies in LLM-generated code. The taxonomy contains 24 inconsistency types, categorized into five dimensions, i.e., *Formatting Inconsistency*, *Semantic Inconsistency*, *Expression/Statement Inconsistency*, *Control Follow Inconsistency*, and *Fault Tolerance Inconsistency*. ❷ Our analysis reveals significant style inconsistencies between human-written code and all studied LLMs, particularly in statements/expressions and formatting. Coding styles among the LLMs themselves are generally similar, with some variation in formatting. ❸ Overall, LLM-generated code is comparable to human-written code in terms of readability, conciseness, and robustness. ❹ Certain prompt strategies can marginally improve the readability and robustness of generated code, but there is a trade-off with conciseness. This suggests that while prompt engineering can help, it may not fully resolve coding style issues and can sometimes reduce code accuracy. ❺ Our case studies show that LLM-generated code often exhibits poor formatting,

[1] In this paper, we use "coding style inconsistency", "style inconsistency", and "inconsistency" interchangeably.

lacks familiarity with basic Python functions, and underutilizes advanced syntax features, making the code less readable, concise, and efficient.

We summarize the main contributions of this paper as follows:

- We provide a comprehensive taxonomy of coding style inconsistencies between LLMs and human developers. Compared to previous work, our proposed taxonomy achieves a more detailed classification of coding styles.
- We conduct extensive analysis to reveal the coding style inconsistencies of several mainstream LLMs, leading to a deeper understanding of LLM-based code generation.
- We propose some practical solutions to improve the inconsistencies of the coding style, paving the way for a more harmonious integration of LLM and coding practices.

2 Related Work

2.1 LLM-Based Code Generation

Code LLMs, such as StarCoder [39], CodeLlama [60], and DeepSeek-Coder [18], are specifically optimized for code-centric tasks [92, 94], leveraging massive code-specific corpora and specialized training instructions. The most state-of-the-art general-purpose large model, GPT-4, due to its significantly larger number of parameters compared to other Code LLMs, demonstrates outstanding performance on code-centric tasks. In recent years, some works have studied the application of LLMs in fields such as vulnerability detection [12, 68, 75, 79, 84], commit message generation [46, 47, 69, 91], unit test generation [61, 63, 77, 83], code search [17, 22, 30, 42, 72], code summarization [3, 21, 37, 65, 66, 73] and code generation [20, 24, 25, 36, 41, 43, 45, 51, 67, 70, 71, 74, 76, 82, 95], etc.

To understand the code generation performance of LLMs, some high-quality code generation benchmarks have been proposed in recent years. For example, HumanEval [10], MBPP [4] and ClassEval [16], covering different scenarios such as repository-level code generation [34, 35, 58, 87] and class-level code generation tasks [16]. While most studies are primarily concerned with improving the functional correctness of code generated by models, using metrics like passk [10], recent research has begun to explore other attributes of code generated by LLMs. For instance, methods have been proposed to enhance the robustness of LLMs [11, 88], and attention has been given to the security aspects of LLMs in code generation tasks, investigating potential vulnerabilities and risks [14, 53].

In contrast to previous works, our investigation is on the code style of LLMs. We conduct the first study to compare the code style of several mainstream LLMs with code written by human programmers. Additionally, we compare the code styles among different mainstream LLMs. This analysis provides insights into the strengths and weaknesses of LLMs in terms of coding style, shedding light on potential areas for improvement and future research directions.

2.2 Coding Style

In previous work [55], Oman et al. proposed a programming style taxonomy, encompassing typographic style, control structure style, and information structure style, which laid the foundation for the development of programming style guidelines and analyzers. However, certain rules in this taxonomy may now be considered outdated and may not fully reflect modern programming practices and conventions (For instance, the example rules mentioning the use of Goto statements are no longer widely used in contemporary programming). Recent strides in coding style research include innovations like CODEBUFF [57], an automatic code formatter that leverages machine learning to understand and apply code formatting styles. Similarly, STYLE-ANALYZER [48] addresses code formatting inconsistencies using a decision tree forest model. However, both CODEBUFF and STYLE-ANALYZER focus solely on formatting style. Mi et al. [49] employed hierarchical

agglomerative clustering to gauge code style inconsistencies, focusing on C/C++ languages. In a recent study [9], DUETCS extracted comprehensive code style features from target code examples, covering text and structure style elements. DUETCS utilizes a Siamese feature network to transform source code style into that of target examples while preserving semantic integrity.

In our study, we adopt the definition of code style provided by earlier works [7, 49, 55, 80]: Code style can be understood as a personal writing habit in source code, reflecting individual preferences in aspects like physical layout, algorithms, and more. While previous research offers a generalized definition of code style, the classification of code style lacks sufficient detail and comprehensiveness. To address this gap, we conducted open coding on code samples generated by several mainstream LLMs. This process result a coding style inconsistency taxonomy comprising 5 dimensions and 24 distinct inconsistency types, offering a comprehensive classification for code style. In comparison to prior efforts, our proposed taxonomy of code style inconsistencies is more comprehensive and detailed, extending beyond traditional focus on text style and structure style. Moreover, our study lays the groundwork for future research on the coding style of LLMs, offering valuable insights and avenues for further exploration in this domain.

3 Experimental Setup

In this section, we introduce the experimental setup, including the LLM selection, benchmark selection, and implementation details.

3.1 LLM Selection

We select five LLMs for our experiments, including four well-known Code LLMs (CodeLlama-7B [60], StarCoder2-7B [39], DeepSeekCoder-1.3B [19], and DeepSeekCoder-6.7B [19]) and the state-of-the-art general-purpose model, GPT-4 [2]. The rationale behind choosing these four Code LLMs is their demonstrated strong performance in code generation tasks. As for GPT-4, it is chosen due to its status as the most effective and parameter-intensive model. The four selected code LLMs are base models without instruction-tuning, making them particularly well-suited for our code generation scenario, where the objective is to generate code based on the provided context.

3.2 Benchmark Selection

Our experiments are conducted on **CoderEval** [81], which is a benchmark used to evaluate code generation performance on pragmatic code generation tasks, i.e., code generation with repository context. It consists of 230 Python and 230 Java tasks from real-world open-source projects. Each task contains a function signature[2], a task description, a solution code as the ground truth, and several unit tests to assess the functional correctness of the generated code. **It is worth noting that the repositories to which these Python tasks belong do not have specific requirements for code style.** The objective of each task is to complete the code specified by the function signature, guided by the provided task description, and ensure that it passes the associated unit tests. In this study, we focus on Python tasks due to Python's popularity [64] and its alignment with previous code generation work [5].

3.3 Implementation Details

During the inference stage, for the four Code LLMs, we employ a random sampling strategy, setting the maximum context length to 1,024 and the temperature to 0.8. For GPT-4, we adopted identical hyperparameter settings. The choices for maximum context length and temperature are aligned

[2]We use "method" and "function" interchangeably in this paper.

with the experimental setup used in CoderEval [81]. All experiments are conducted on a machine with 216 GB of RAM and a Tesla A100 GPU with 80 GB of memory.

4 Evaluation

In this section, we report and analyze the experimental results to answer the following research questions (RQs):

- **RQ1:** What are the types of coding style inconsistencies between the selected LLMs and human?
- **RQ2:** What is the distribution of the coding style inconsistencies?
  - **RQ2.a**: What are the percentages of inconsistent coding style for different models?
  - **RQ2.b**: What are the inconsistency type numbers present between a single code sample and the corresponding ground truth?
  - **RQ2.c**: What are the distribution of coding style inconsistency types for models?
- **RQ3:** How do model-generated code and human-written code compare in three aspects: readability, conciseness, and robustness?
- **RQ4:** Can prompting techniques improve the coding style of LLMs?

4.1 RQ1: Coding Style Inconsistency Identification

To identify the inconsistencies in coding styles between LLMs and human programmers, we manually analyze the generated results of the five LLMs. It's worth noting that human-written code does not always equate to good coding style. **Therefore, we do not use human-written code as a standard to assess the quality of coding style in LLM-generated code. Instead, we focus solely on analyzing the coding style inconsistencies between human-written code and LLM-generated code.** By comparing these generated results with the ground truths, we summarize the types of coding style inconsistencies. We conduct open coding [29] on the code generated by LLMs. Initially, we describe the data collection process, followed by a detailed explanation of the coding protocol.

4.1.1 Data Collection. Our data collection process includes three steps: model generation, automatic filtering, and manual filtering.

Model generation. For each of the 230 Python code generation tasks from CoderEval [81], we prompt the five LLMs to perform code generation using the same prompting template. For each task, we instruct each model to generate 10 results, resulting in an initial total of 2,300 code samples for each model.

Automatic filtering. To ensure the correctness of the collected code samples, we further filter out code samples that fail to pass any of the associated unit tests for the task, leading to 456, 189, 365, 497 and 570 results that pass all tests for five LLMs, respectively. We further merge identical code samples to reduce analysis effort, resulting in 1,557 unique code samples. We only annotate these unique code samples to ensure that the annotation results for the same code sample generated by different models are consistent, thereby avoiding the situation where the same code sample generated by different LLMs is annotated with different results.

Manual filtering. To ensure the quality of collected code samples, we manually check and filter them based on three criteria: ❶ *Style consistency*. We filter out results that exhibit no inconsistency in coding style. If the naming conventions, commenting style, code structure, and other aspects of the two code samples are consistent, we conclude that there is no significant difference in code style between them. For example, Figure 1 shows an example of consistent coding style between the code sample generated by Code LLM and corresponding ground truth. As a result, 73 code samples are filtered out in this way. ❷ *Functional correctness*. We filter out results that implement the task

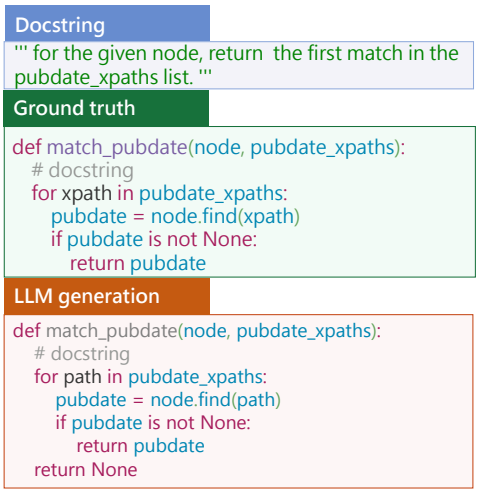


Fig. 1. An Example of Style-Consistent Implementation.

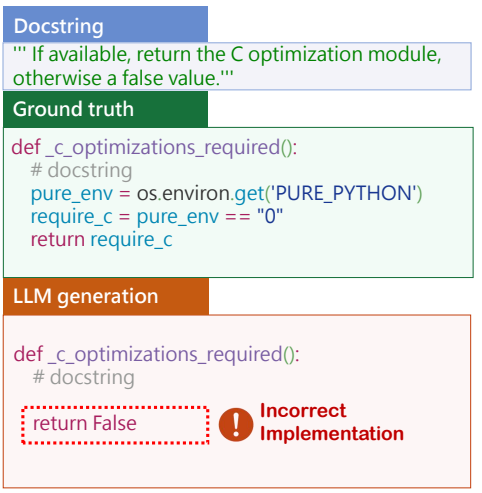


Fig. 2. An Example of Incorrect Implementation that Passed Unit Tests.

incorrectly despite passing the unit tests. Previous work has shown that existing benchmarks suffer from test sufficiency issues, meaning that even if a generated result passes all tests, there is still a chance it could be incorrect [44]. For example, Figure 2 shows an example of wrong implementation although passing test cases. As a result, 286 code samples are filtered out in this way. ❸ *Implementation conciseness.* We filter out results that contain extra code that does not contribute to fulfilling the function's implementation requirements (e.g., two exactly the same loops). As a result, 19 code samples are filtered out in this way. Finally, we obtain 1,179 unique code samples for the study.

4.1.2 Data Annotation. We adopt classifications of coding style inconsistencies in previous work [9] as the initialization of our classification and conduct open coding [29] on the code samples generated by LLMs. Our objective is to refine and expand these classifications to capture detailed instances of coding style inconsistencies. **In our annotation process, three annotators were involved, each with over five years of Python programming experience and a solid understanding of the field of code style.**

Iterative coding. The three annotators independently analyze the code samples one by one. For each code sample, the annotators independently compare it with the ground truth line by line to identify inconsistencies, without knowing which model produced the result. If a code sample and its corresponding ground truth show inconsistency that matches a current definition of inconsistency type, the annotators code the generated result with the specific inconsistency type. If the inconsistency does not fit any existing definitions, the annotators either modify an existing definition or create a new type. When the inconsistency types are updated, all code samples will be re-annotated to ensure consistency. Note that a code sample can be classified under multiple inconsistency types. For example, if a code sample follows a different naming convention (Naming Formatting Inconsistency) and also structures loops differently (Loop Structure Inconsistency), it will be annotated with both inconsistency types.

This iterative coding process aims to capture the nuanced nature of coding style inconsistencies. During the coding process, the annotators also summarize guidelines for each inconsistency type

annotation to ensure clarity and consistency in our annotations. These guidelines include specific examples and detailed descriptions to help identify and classify each type of inconsistency accurately. This ensures the annotation consistency and the reproducibility across different coders.

Periodic review and update. After analyzing every 50 code samples, the three annotators conduct a review of both the taxonomy and the coded samples. Based on insights from the review and discussions, we refine the definitions of inconsistency types, merging or removing types as necessary. Following any updates to the taxonomy, all code samples are re-annotated to maintain consistency and accuracy in the categorization of inconsistencies. This periodic review and update process continues until all code samples have been fully coded, ensuring thorough and reliable identification of coding style inconsistencies. Note that the taxonomy has remained stable during the last several reviews, indicating a mature and robust classification system. We calculated that the average disagreement rate among the three annotators was 37.6% after labeling every 50 samples. However, these disagreements were resolved through discussions among the annotators.

4.1.3 Taxonomy. Table 1 presents the 24 inconsistency types identified during the open coding, along with their names and definitions. For each inconsistency type, the full annotation results and detailed annotation guidelines are included in our replication package [1].

Taxonomy Analysis. We have further categorized the 24 types of inconsistencies into five dimensions based on their main focus:

- **Formatting Inconsistency**. This dimension focuses on inconsistencies related to code formatting, such as indentation, spacing, and code/comment layout.
- **Semantic Inconsistency**. This dimension focuses on inconsistencies related to the meaning or semantics of code, including variable naming, function naming, and the level of detail in comment style.
- **Expression/Statement Inconsistency**. This dimension focuses on inconsistencies related to the style or usage of expressions and statements within the code, such as assignment styles, conditional expressions, and data structure construction.
- **Control Follow Inconsistency**. This dimension focuses on inconsistencies related to control flow structures within the code, such as conditional statements, loop structures, and exception handling.
- **Fault Tolerance Inconsistency**. This dimension focuses on inconsistencies related to error handling and fault tolerance mechanisms within the code, including input validation, runtime validation, and exception handling.

It is worth noting that we have categorized 24 inconsistency types into 5 dimensions, with the scope of code style inconsistencies contained in each dimension solely determined by the inconsistency types assigned to that dimension. Since there is no obvious overlap between the inconsistency types, there is also no clear overlap between the 5 dimensions.

Figure 3 provides a visual representation of the relationships between the five dimensions and the 24 inconsistency types identified. The inconsistency types are organized into a tree-like structure in the figure, with the dimensions and inconsistency types represented using different shapes, connected by lines. Furthermore, these inconsistencies vary in their scopes of influence, such as identifier, statement, and block, as also depicted in Figure 3. Some inconsistencies may belong to only one or a few identifiers (e.g., Naming Formatting Inconsistency) or a single statement (e.g., Assignment Style Inconsistency), while others may impact an entire block of code (e.g., Loop Structure Inconsistency) or span across multiple blocks (e.g., Code Order Inconsistency).

Table 1. Coding Style Inconsistency taxonomy.

ID	Inconsistency Type	Definition
1	Naming Format Inconsistency	Inconsistencies in the formatting of identifiers (e.g., variable names, function names, or parameter names), such as using camelCase (e.g., authorName) versus snake_case (e.g., author_name).
2	Space Inconsistency	Inconsistencies in the use of space(e.g., whitespace and indentation) around various syntactical elements, e.g., operators, colons, comments, and brackets.
3	Blank Line Inconsistency	Inconsistencies in the use of blank lines. For example, one style includes blank lines to separate code blocks, while the other omits them.
4	Inline Code Usage Inconsistency	Inconsistencies in the usage of inline code constructs. It encompasses cases where one approach employs inline expressions or functions while the other does not.
5	Comment Format Inconsistency	Inconsistencies in the formatting of comments within code. It includes variations in interline comments, inline comments, commented-out code, and trailing comments.
6	Statement Organization Inconsistency	Inconsistency in the organization style of statements, exemplified by completing expressions or statements in a single line in contrast to breaking them into multiple shorter lines.
7	Naming Semantics Inconsistency	Inconsistencies in the semantic meaning of identifiers, such as using generic single-letter identifiers (e.g., i, l, d) versus meaningful, descriptive words (e.g., index, length, day).
8	Comment Semantics Inconsistency	Inconsistencies in the semantic aspects of comments within code, such as variations in the level of detail or semantic differences, or with TODO comment, useless comments.
9	Assignment Inconsistency	Inconsistencies in the style of variable assignment, e.g., tuple unpacking assignment, chained assignment, separate assignment. Examples include using augmented assignment versus standard assignment (e.g., ‘x += 1‘ vs. ‘x = x + 1‘).
10	Conditional Syntax Inconsistency	Inconsistencies in the syntax used for conditional statements within code. It covers scenarios where one method involves conditional statements while the other employs conditional expressions or return statements with equivalent functionality.
11	Conditional Expression Inconsistency	Inconsistencies in the way conditional expressions are written, despite having similar functionalities. For example, one style might use if len(a) > 1 while another uses if len(a) >= 2.
12	Data Structure Construction Inconsistency	Inconsistencies in the methods used to construct data structures such as lists, dictionaries, sets, tuples, strings, and iterators. For example, using different syntaxes or functions to create these data structures.
13	API Preference Inconsistency	Inconsistencies in how APIs are used to achieve similar functionality. It includes variations such as calling different functions or methods defined in the repository, using built-in functions, or re-implementing the functionality without calling existing functions.
14	Advanced Syntax Usage Inconsistency	Inconsistencies in the use of advanced syntax features, such as lambda expressions
15	Code Ordering Inconsistency	Inconsistencies in the order of semantically similar code blocks, such as import statements, assignments, loops, and other logical sections of code.
16	Loop Structure Inconsistency	Inconsistencies in loop structures within code. It covers scenarios where one approach employs a for loop while the other uses a while loop, or where one loop contains only a basic loop structure while the other includes additional control flow statements such as if-break, for-else structure, and while-else structure
17	Conditional Structure Inconsistency	Inconsistencies in the structure and design of conditional statements within methods. It includes variances such as the use of multiple conditional statements versus a single statement with equivalent meaning, differences in the structures of multiple conditional statements while preserving the same semantics, and disparities in the inclusion of return statements alongside conditional statements.
18	Control Flow Structure Inconsistency	Inconsistencies in the use of control flow structures, such as using if-else versus try-except statements during the code execution (e.g., input and runtime validation).
19	Input Validation Presence Inconsistency	Inconsistencies in whether input checking with conditionals is performed.
20	Runtime Validation Presence Inconsistency	Inconsistencies in whether runtime validation with conditionals is performed, ensuring data integrity during code execution..
21	Exception Handling Presence Inconsistency	Inconsistencies in whether exceptions are handled, e.g., using try-except blocks, to manage errors that occur during execution.
22	Input Validation Style Inconsistency	Inconsistencies in the style of input validation with conditionals (ensuring input data is checked before processing), such as whether exceptions are thrown, the types of exceptions used, and the use of logging.
23	Runtime Validation Style Inconsistency	Inconsistencies in the style of runtime validation with conditionals during code execution, such as whether exceptions are thrown, the types of exceptions used, and the use of logging.
24	Exception Handling Style Inconsistency	Inconsistencies in the style of exceptions that occur during execution are handled, such as whether exceptions are thrown, the types of exceptions used, and the use of logging, the use of try-else block.

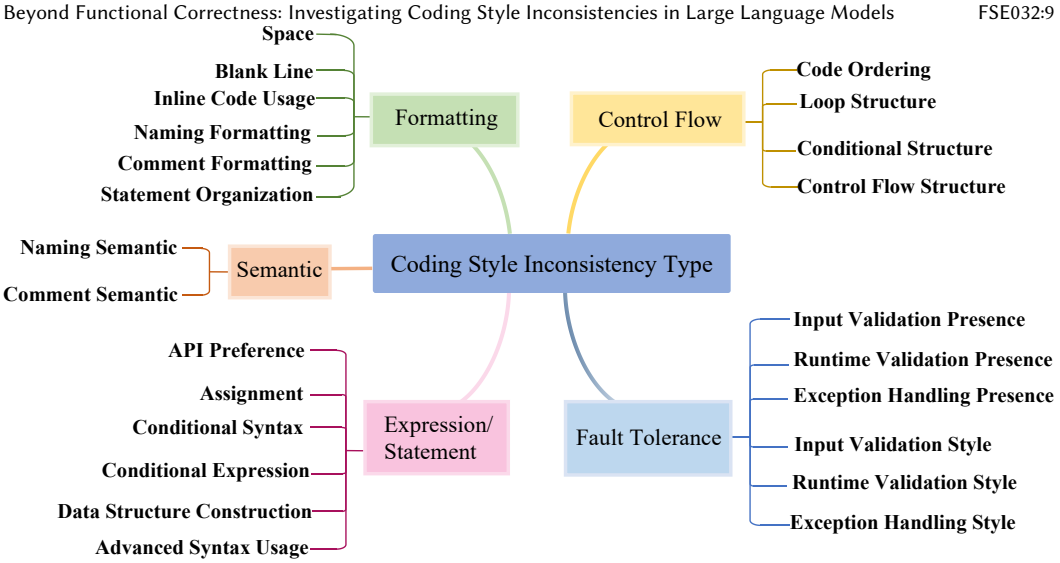


Fig. 3. Dimensions and Corresponding Coding Style Inconsistency Types.

Note that certain inconsistencies could affect both statement and block structures, contingent upon the complexity of the code involved. For instance, in the context of API preference inconsistency, the implementation of the same functionality may vary. It could involve calling different single APIs within a statement, or it might require the coordination of several APIs with specific usage patterns across multiple code blocks.

Taxonomy Comparison. Compared with the coding style taxonomy of Chen et al. [9], they categorize coding styles into text style and structure style, with four subtypes formatting, naming, ordering of code blocks, and control structures. Our taxonomy covers all these types and introduces three additional dimensions: semantic, expression/statement, and fault tolerance. We expand upon their framework by introducing 24 fine-grained types compared to 4 types. For instance, we refine their subtype Control Structures into three specific inconsistency types related to: Conditional Structure Inconsistency, Loop Structure Inconsistency, and Control Flow Structure Inconsistency, offering a more detailed classification. Our taxonomy is backed by comprehensive guidelines derived from actual open coding, providing detailed and actionable classifications.

Taxonomy Generalizability. One concern is that, since each task in CoderEval has only one ground truth, this may reflect the coding style of a single programmer, potentially limiting the generalizability of our taxonomy. To this end, we randomly selecting 42 Python tasks from the 230 available in CoderEval and two annotators were assigned to independently rewrite the ground truths for these 42 tasks, resulting in a total of 84 ground truths. Another annotator was responsible for verifying the correctness of these ground truths. We then identified the code samples corresponding to these 42 tasks from a set of 1,179 unique code samples and annotated the coding style differences between the rewritten ground truths and the code samples following the outlined annotation process. Both the rewritten ground truths and the annotation results are available in our replication package [1]. After the annotation process, the overall taxonomy remained unchanged.

In summary, our taxonomy not only complements but also substantially enhances previous research, filling critical gaps and offering a more robust framework for analyzing the inconsistencies in coding style. Note that while our taxonomy is based on summarizing inconsistencies observed in Python code generated by LLMs, it is not limited to Python alone. The concepts and categories can be generalized to other programming languages as needed.

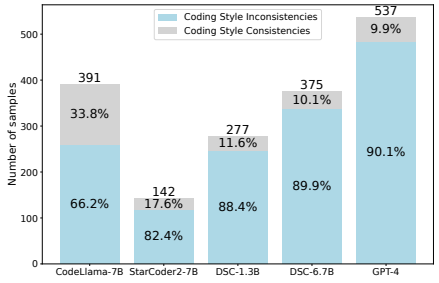


Fig. 4. Percentages of Inconsistent Coding Styles.

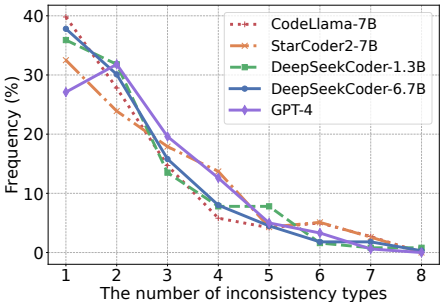


Fig. 5. Inconsistency Numbers in a Single Code Sample.

RQ1 Summary: We have identified 24 types of coding style inconsistencies and categorized them into five dimensions: Formatting, Semantic, Expression/Statement, Control Flow, and Fault Tolerance. Our taxonomy expands upon previous work by introducing new dimensions and providing more detailed classifications with guidelines.

4.2 RQ2: Coding Style Inconsistency Analysis

We design RQ2 to evaluate the differences between human-written code and LLM-generated code. Specifically, we investigate the coding style differences in three perspectives: (1) Percentages of inconsistent coding styles; (2) Inconsistency numbers present in a single code sample; and (3) Distribution of coding style inconsistency types.

4.2.1 Percentages of Inconsistent Coding Styles. We counted the number of code samples generated by the five models (CodeLlama-7B, StarCoder2-7B, DeepSeekCoder-1.3B, DeepSeekCoder-6.7B, and GPT-4) that passed unit tests, were manually verified for functional correctness, and did not contain extra code. The results were 391, 142, 277, 375, and 537, respectively. Furthermore, we calculated the proportion of these code samples that either exhibit or do not exhibit coding style inconsistencies compared to human-written code. The statistical results are shown in Figure 4. From Figure 4, we can find that all LLMs exhibit coding style inconsistency with human-written code and the degree varies: 66.2%, 82.4%, 88.5%, 89.9% and 90.1% for the five models, respectively.

4.2.2 Inconsistency Numbers Present in a Single Code Sample. For each model, we counted the number of inconsistency types present in each code sample (considering only the code samples that exhibit coding style differences with the corresponding ground truths). Then, We counted the frequency of different numbers of inconsistent types in one sample for each model. A line chart was plotted based on the frequency of inconsistency types present in the code samples. From Figure 5, it can be seen that the number of inconsistent types for one code sample ranges between 1 and 8. For each model, the trend of the frequency line chart is roughly the same, with all lines generally showing a decreasing trend. The code samples of CodeLlama-7B, StarCoder2-7B, DeepSeekCoder-1.3B, and DeepSeekCoder-6.7B show the highest frequency of having one inconsistency type, at 39.8%, 32.5%, 35.9%, and 37.8%, respectively. The code samples of GPT-4 exhibit the highest frequency of two inconsistency types, at 31.8%. The lowest frequency is that code samples with 8 inconsistency types, at 0.0%, 0.0%, 0.8%, 0.3% and 0.0%, respectively.

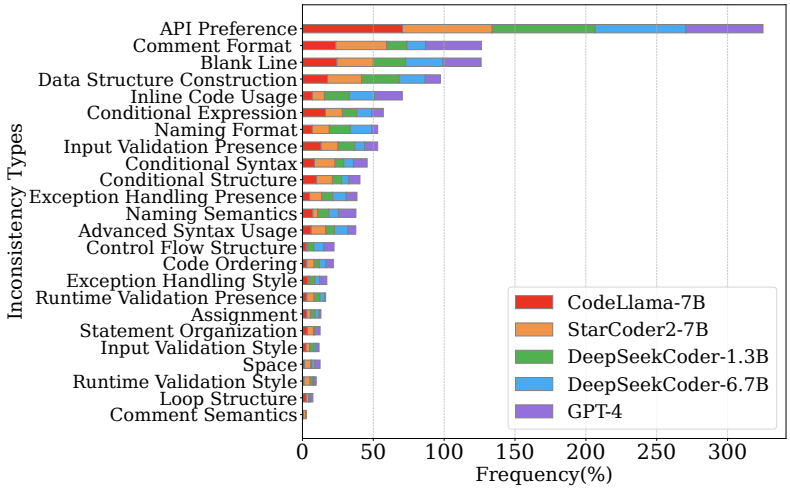


Fig. 6. Overall Distribution of Coding Style Inconsistency Types.

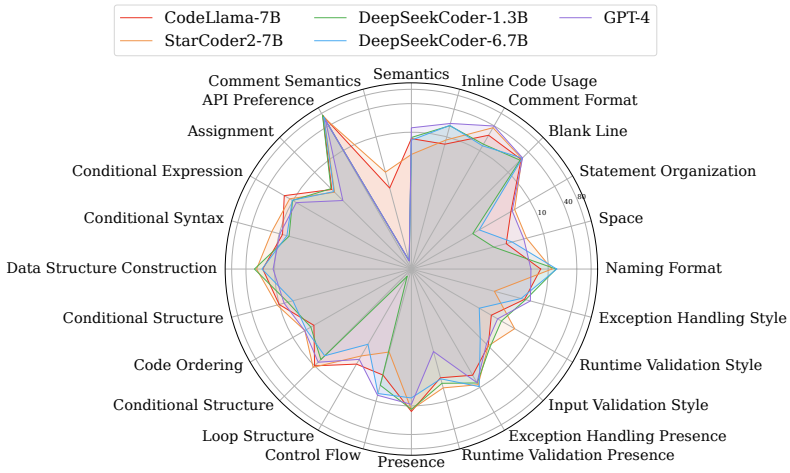


Fig. 7. Breakdown Distribution of Coding Style Inconsistency (by Model). The "Semantics" at the top refers to "Naming Semantics", and the "Presence"at the bottom refers to "Input Validation Presence".

4.2.3 Distribution of Coding Style Inconsistency Types. Figure 6 is a stacked plot showing the frequencies of a certain inconsistency type in code samples generated by five models. For example, the frequencies of API Preference inconsistency in the code samples generated by CodeLlama-7B, StarCoder2-7B, DeepSeekCoder-1.3B, DeepSeekCoder-6.7B, and GPT-4 are 70.7%, 63.3%, 72.7%, 64.1%, and 54.4%, respectively, summing up to 325.0%. We can observe that the top-4 inconsistency types are API Preference Inconsistency (325.0%), Comment Formatting Inconsistency (126.3%), Blank Line Inconsistency (126.1%) and Data Structure Construction Inconsistency (97.4%). Among these top four inconsistency types, API Preference Inconsistency stands out with a significantly higher frequency, even surpassing the combined frequencies of the second and third-ranked types. In contrast, the bottom inconsistency types are: Runtime Validation Style (9.7%), Loop Structure

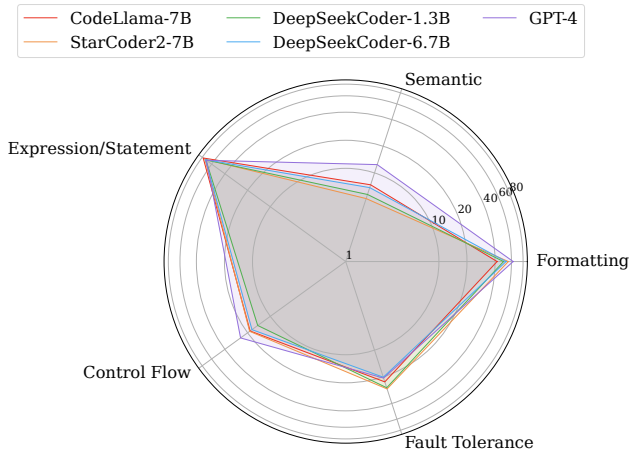


Fig. 8. Breakdown Distribution of Coding Style Inconsistency (by Dimension).

(7.4%), Comment Semantic (2.5%). The low frequencies of these types indicate that LLMs and human-written code are relatively consistent in these aspects.

In order to understand the inconsistencies deeper, we conducted a detailed analysis of the top-4 inconsistency types. In our observed code samples and the ground truths, we found that the code samples and the ground truths might call functions from different sources to achieve similar functionality. Different sources refer to functions that may be defined within the original repository, built-in Python functions, etc. For example, we found that in 6.6% of cases, the ground truth calls functions defined in the original repository while similar functionality is achieved using Python built-in functions, etc., in the code samples generated by models. This may be because the model lacks contextual information about the functions defined in the original repository when generating code. As a result, the large model uses built-in functions or third-party library functions, etc., to achieve similar functionality. For instance, in one task, the ground truth uses a function defined in the original repository, *"match_file_by_prefix(prefix, file)"*, to check if the prefix of the file name is "prefix", while the code sample generated by models uses the built-in method in Python *"startswith"* to achieve similar functionality.

Comment Format Inconsistency and Blank Lines Inconsistency are the second and third most frequent inconsistency types. In our analysis, we found that model-generated code samples tend to avoid using blank lines to separate code blocks compared to human-written code. The five models generally have a high frequency of Comment Formatting inconsistency, but there are differences among them (The highest frequency is observed for GPT-4 at 39.5%. StarCoder2-7B follows with 35.9%, and CodeLlama-7B ranks third at 23.6%. DeepSeekCoder-1.3B and DeepSeekCoder-6.7B have the lowest frequencies, recorded at 14.3% and 13.1%, respectively.). Among these models, GPT-4 has the highest frequency, partly because it includes more meaningful block and inline comments compared to human-written code. The other four models are less inclined to add such comments. Notably, code samples from CodeLlama-7B, StarCoder2-7B, and DeepSeekCoder-1.3B even do not include any examples containing inline comments. The relatively high frequency of StarCoder2-7B and CodeLlama-7B can be attributed to their less standardized comment formatting compared to the other models. For example, the code samples generated by CodeLlama-7B and StarCoder2-7B may contain commented-out code or TODO comments, while the code samples generated by

DeepSeekCoder-1.3B, DeepSeekCoder-6.7B and GPT-4 do not. We consider that having commented-out code in code is not good coding practice because these comments are unnecessary information and do not help in understanding the functionality of the code. We believe that including TODO comments in code generated by large models is not good coding practice. This is because we expect large models to produce complete code based on requirements, rather than including comments indicating unfinished tasks or future improvements. Data structure construction inconsistency is a frequently occurring type of inconsistency. The code samples and the corresponding ground truths may show differences in constructing data structures (e.g., list, set). In our observed samples, human programmers tend to prefer using list comprehensions to construct lists, whereas the code samples generated by LLMs tends to favor conventional methods for constructing lists.

Figure 7 shows a radar chart of the frequency of inconsistency types for five different models, allowing us to compare the overall frequency distribution of inconsistency types across these models. As shown in Figure 7, the distribution of inconsistency types for DeepSeekCoder-1.3B and DeepSeekCoder-6.7B is relatively similar compared to the other models in terms of inconsistency types such as comment format, conditional expression, and naming format, etc. For example, in the Inline Code Usage inconsistency type, the frequency for DeepSeekCoder-1.3B and DeepSeekCoder-6.7B is very close, lower than the frequency for GPT-4, but higher than that for CodeLlama-7B and StarCoder2-7B. The frequencies of DeepSeekCoder-1.3B, DeepSeekCoder-6.7B, and GPT-4 are higher because they tend to include more intermediate variables in the code compared to the ground truths. Therefore, we can conclude that the base model significantly influences the coding style. The training data and method have a more noticeable impact on the coding style of the model compared to the parameters.

Figure 8 presents a radar chart that summarizes coding style inconsistencies by grouping them into five broader dimensions, i.e., formatting, semantic, expression/statement, control flow, and fault tolerance. To calculate the frequency for each dimension, we sum the instances of inconsistency types belonging to that dimension and divide it by the total number of valid code samples. From Figure 8, we have the following observations:

- It is evident that the coding styles of different LLMs are similar in dimension granularity. This is indicated by the almost overlapping shapes on the radar chart, highlighting that these models share a similar distribution of inconsistency types by dimension.
- The dimensions, ranked by average frequency of inconsistencies, are as follows: statement/expression (73.1%), formatting (52.4%), fault tolerance (24.3%), control flow (18.0%), and semantic(7.5%). The high ranking of statement/expression inconsistency is primarily due to the significantly high frequency of API Preference Inconsistency within this dimension.
- We then calculate the difference between the highest and lowest values of frequency of inconsistencies for each dimension. We sort the five dimensions from high to low according to the difference, and the result is: formatting (20.5%), semantic (7.3%), statement/expression (7.2%), fault tolerance (7.2%), and control flow (6.0%). This is because, although the training data of the models is generally similar, there are still some differences.

RQ2 Summary: There are obvious coding style inconsistencies between human and all the studied LLMs. The top inconsistency type is API preference and top inconsistency dimensions are statements/expressions and formatting dimensions. While LLMs generally have similar coding styles, there are also noticeable differences in the formatting dimension.

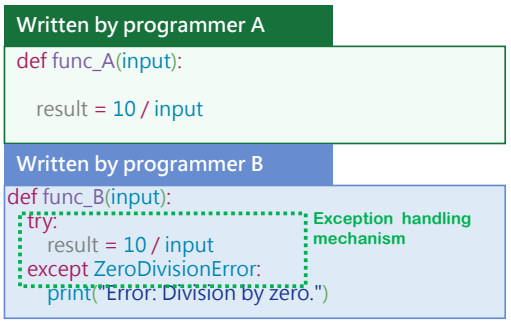


Fig. 9. An example of how robustness relates to code style.

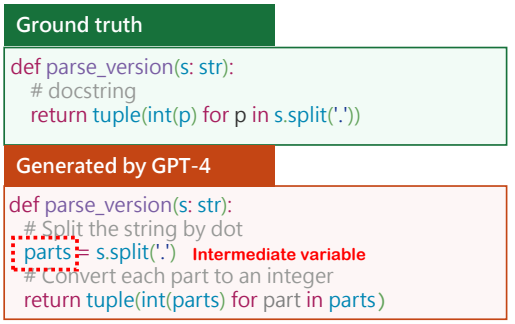


Fig. 10. An example of generated code being less concise than Human-Written code.

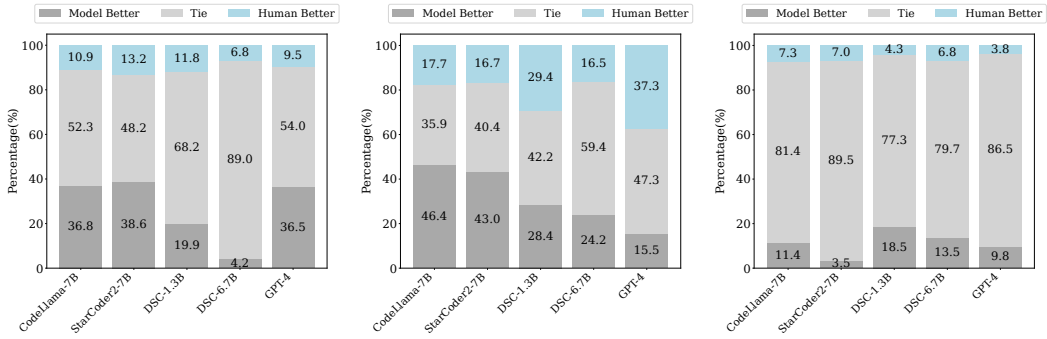
4.3 RQ3: Coding Style Comparison

In addition to analyzing coding style inconsistency between LLMs and human programmers, we further compare the code samples generated by LLMs and the ground truths in terms of three aspects: readability, conciseness, and robustness.

- Readability: The readability and understandability of code.
- Conciseness: The simplicity of the code and the degree to which it is free of unnecessary elements.
- Robustness: The ability of the code to handle corner cases and potential errors. To illustrate the relationship between robustness and code style, consider two programmers, A and B. Programmer B prefers to implement exception handling mechanisms in the code, while Programmer A tends to overlook this practice. In the Figure 9, `func_A` works correctly when the input is valid, but it will crash if the input is zero. In contrast, `func_B` includes an exception handling mechanism, ensuring the program does not crash and providing a meaningful error message when division by zero occurs. This demonstrates how Programmer B's code is more robust due to different coding style.

We compare the code samples generated by LLMs from RQ1 with their corresponding ground truths and evaluate each of the three aspects. For each analysis, we determine whether the generated code is better than the ground truth ("model better"), whether they are comparable ("tie"), or whether the ground truth is better ("human better"). The scoring for the three aspects follows these principles: (i) Readability: If code sample A conveys the intent and logical structure more clearly than code sample B, it is superior in readability. (ii) Conciseness: If code sample A achieves the same functionality as code sample B while eliminating unnecessary redundancies, it is considered superior in terms of conciseness. This involves reducing the number of intermediate variables, optimizing loops, etc. (iii) Robustness: If code sample A effectively implements error handling and fault tolerance mechanisms, whereas code sample B lacks such mechanisms, the former is deemed superior in robustness. This includes the presence of input validation, exception handling, etc.

The annotation is conducted independently by two annotators. Both annotators have many years of programming experience and possess a certain level of understanding regarding research on code style. The two annotators independently score the code samples, resulting in two sets of annotations. Cohen's Kappa [13] is a statistical measure used to assess the consistency between two annotators for a classification task. We calculated the Cohen's Kappa values for the two annotations on readability, conciseness, and robustness, which are 0.63, 0.79, and 0.68, respectively. However,



(a) Readability (b) Conciseness (c) Robustness

Fig. 11. Score Distribution across Readability, Conciseness, and Robustness.

the discrepancies that arose in the annotations were ultimately resolved through discussions to reach a consensus.

Figure 11 show the proportion of code samples that received different scores (model better, tie and human better) on the three aspects for each model. Overall, the code samples generated by the LLMs is comparable to that written by human programmers in the three aspects. On average, the code generated by the five models is comparable to or even superior to the code written by programmers in 89.5%, 76.5%, and 94.2% of cases in terms of readability, conciseness, and robustness, respectively. The following is a comparative analysis of the readability, conciseness, and robustness of the code samples generated by the five LLMs. From the perspective of readability, the code samples generated by DeepSeekCoder-6.7B have the highest readability, while those generated by StarCoder2-7B have the lowest. In terms of conciseness, the conciseness of code samples generated by CodeLlama-7B, StarCoder2-7B, and DeepSeekCoder-6.7B is comparable, while GPT-4 generates less concise code. Figure 10 presents an example that the conciseness of a code sample generated by GPT-4 is inferior to that of ground truth written by human programmers. Note that conciseness and readability are often trade-offs. As shown in Figure 10, GPT-4 enhances readability by splitting one statement into two lines. All five studied LLMs demonstrate relatively high robustness. This suggests that the models might have learned more robust coding styles from their training data, such as more rigorous input parameter checks, which human programmers might omit due to oversight or to avoid excessive complexity.

> **RQ3 Summary:** Overall, the code samples generated by the five LLMs is comparable to human-written code in terms of readability, conciseness, and robustness. Among the studied models, DeepSeekCoder-6.7B produces the most readable code, while code samples generated by GPT-4 has the lowest simplicity but the highest robustness.

4.4 RQ4: Style Improvement by Prompting Techniques

In this RQ, we investigate whether prompting techniques can improve the coding style of LLMs. We conduct experiments with DeepSeekCoder-6.7B on 20 sampled Python tasks from CoderEval. These tasks are randomly selected from those that DeepSeekCoder-6.7B can complete, meaning DeepSeekCoder-6.7B can generate code samples that pass all corresponding test cases. We design

Prompt-head-concise: # Complete the function below by following the provided function signature and docstring. Ensure the implementation adheres to good coding style practices, including readability, simplicity, and robustness. [Function Signature] [Docstring]

Prompt-head-detailed: # Complete the function below by following the provided function signature and docstring. Ensure the implementation adheres to good coding style practices, including readability, simplicity, and robustness. For example: [Principle 1][Principle 2] [Principle 3] [Function Signature] [Docstring]

Prompt-end-concise: [Function Signature] [The context of initial docstring, Style Guidelines: Complete the function below by following the provided function signature and docstring. Ensure the implementation adheres to good coding style practices, including readability, simplicity, and robustness.]

Prompt-end-detailed: [Function Signature] [The context of initial docstring, Style Guidelines: Complete the function below by following the provided function signature and docstring. Ensure the implementation adheres to good coding style practices, including readability, simplicity, and robustness. For example: [Principle 1][Principle 2] [Principle 3]]

Note:
- Principle 1: Include meaningful inline comments to enhance code readability
- Principle 2: Use if statements for input validation and try-except blocks for exception handling to improve code robustness
- Principle 3: Avoid excessive use of intermediate variables to keep the code simple
- "head" or "end": The placement of the style guidelines
- "concise" or "detailed" :The level of detail of the style guidelines

Fig. 12. Four Enhanced Prompts in RQ4 Study.

four types of enhanced prompts for this study (refer to Figure 12), aiming to instruct the model to generate code with better coding style using explicit style guidelines. The design of these prompts investigates the impact of the placement and detail level of style guidelines. In prompt names, "-head" or "-end" specifies whether the style guidelines are placed before the function signature and docstring, similar to a directive, or appended at the end of the original docstring, simulating a normal docstring style. "-concise" and "-detailed" indicate the level of detail in the style guidelines. The detailed version includes three specific principles related to code readability, conciseness, and robustness, in addition to the concise information.

Among the selected tasks, DeepSeekCoder-6.7B generates 134 functionally correct code samples using the basic prompt, i.e., the original function signature and docstring as input. Then, for each type of enhanced prompt, DeepSeekCoder-6.7B generates 10 code samples for the 20 selected tasks, resulting in 115, 137, 75, and 78 functionally correct code samples for each of the four enhanced prompts, respectively. The accuracy for the four enhanced prompts is 57.5%, 68.5%, 37.5%, and 39.0%, respectively, compared to the 67.0% accuracy of the basic prompt. Except for prompt-head-detailed, the enhanced prompts result in lower accuracy compared to the basic prompt, suggesting that using more complex prompts may lead to a decrease in the functional correctness of the generated code.

The two annotators mentioned in Section 4.3 referred to the scoring principles mentioned in Section 4.3 to score the code samples generated using the basic prompt and the four enhanced prompts for readability, conciseness, and robustness. The Cohen's Kappa for the two sets of annotations on readability, conciseness, and robustness were 0.65, 0.81, and 0.74, respectively. Despite the discrepancies in the annotations, the annotators were eventually resolved through discussions, leading to a consensus. The G-test [59] is a statistical method used to assess the association between categorical variables. In this study, we applied the G-test to determine whether the probability distributions of scores for the basic prompt and the four enhanced prompts show significant differences across the three aspects of readability, conciseness, and robustness. The results indicated that the p-values for all three aspects were below 0.05, suggesting a statistically significant difference in the score distributions among the five prompts across these three aspects.

The results are depicted in Figure 13. Among the enhanced prompts, Prompt-head-concise, Prompt-end-concise, and Prompt-end-detailed slightly improve the readability of the code samples

(a) Readability (b) Conciseness (c) Robustness

Fig. 13. Score Distribution across Readability, Conciseness, and Robustness by Different Prompts. P-1, P-2, P-3, P-4 Stand for Prompt-head-concise, Prompt-head-detailed, Prompt-end-concise, and Prompt-end-detailed, Respectively.

generated by DeepSeekCoder-6.7B. However, as shown in Figure 13(b), only Prompt-head-detailed enhances the conciseness of DeepSeekCoder-6.7B's code samples. This is because there's often a trade-off between readability and conciseness, where improving one may compromise the other. Additionally, as seen in Figure 13(c), all four enhanced prompts contribute to some extent to the improved robustness of DeepSeekCoder-6.7B's code samples. In conclusion: (i) Incorporating style-guiding information into prompts may lead to decreased accuracy in generated code, as observed in our evaluation. (ii) Relying solely on prompt engineering may not fully resolve issues related to code style. Additional strategies or refinements may be necessary.

> **RQ4 Summary:** Certain types of prompts can slightly improve the readability and robustness of generated code, but only one type enhances conciseness. There is a trade-off between readability and conciseness, indicating that while prompt engineering can help, it is not sufficient to fully address issues related to coding style. Including guidance in prompts may also decrease the accuracy of generated code.

4.5 Case Studies

In the code samples we observed, we categorized and analyzed cases where the code samples exhibited inconsistent coding styles compared to the ground truth. We identified the following common scenarios.

Poor formatting characteristics. LLMs may generate code with poor formatting characteristics, such as improper use of spaces. Figure 14 (b) is a function generated by GPT-4. In the expression *"base_name.startswith(prefix+"-")"*, the "+" operator is not surrounded by spaces, which is not a good practice in the use of spaces. The code generated by the other four models also contains improper use of spaces, such as DeepSeekCoder-6.7B generating the code *"yield response[start: end]"*. However, the PEP-8 compliant version should be written as *"yield response[start:end]"*.

Unfamiliar with basic Python features. LLMs might not be very familiar with some basic syntax features, which results in generating more complex code. For example, in Figure 14 (d), DeepSeekCoder-6.7B might not understand list slicing operations well, so it generated more complex code to avoid out-of-bounds indexing. Assuming the list has a length of 4, using `list[3:5]` in Python will not result in an error. Instead, it will return elements from index 3 to the end of the list.

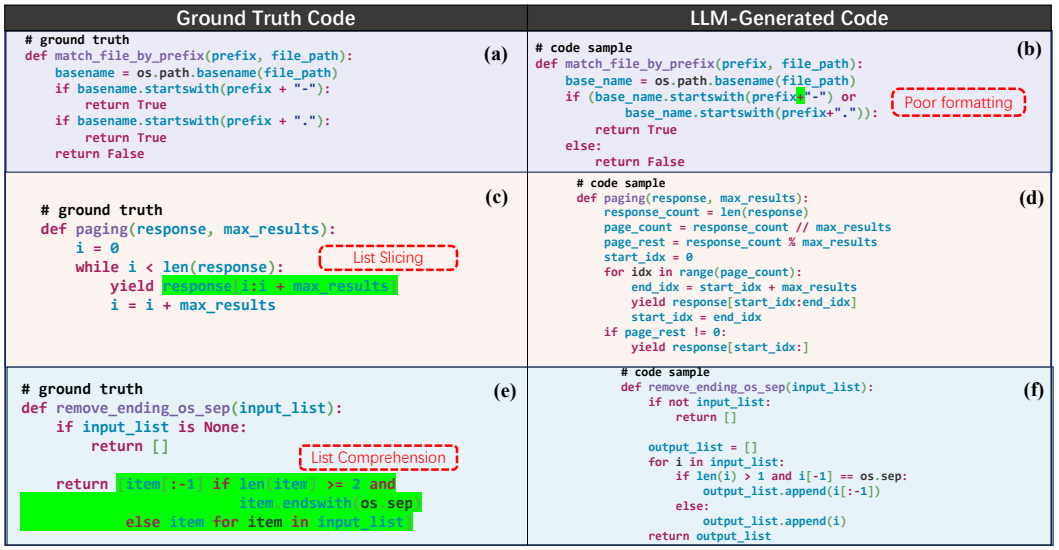


Fig. 14. Examples of (i) Poor formatting characteristics; (ii) Unfamiliar with basic Python features; (iii) Rare use of advanced syntax features. On the left side of the image, (a), (c), and (e) are the ground truth for each example, while on the right side, (b), (d), and (f) are the code samples generated by LLMs.

However, in the corresponding ground truth of the code sample (Figure 14 (c)), the code logic is clear and concise.

Rare use of advanced syntax features. Compared to code written by human programmers, code generated by LLMs often does not use advanced syntax features of the Python language, such as Pythonic idioms. As shown in Figure 14 (e), the ground truth uses list comprehension to build a list, while the code sample in Figure 14 (f), uses a more conventional method to build the list. It first constructs an empty list and then uses the `append()` method to add elements to the empty list. Compared to the ground truth, the simplicity of the code sample is inferior.

5 Threats to Validity

We have identified the following threats to our study.

Data Quality. One potential threat to validity is the quality of the raw data used for our empirical study. To ensure the quality of the data for open coding, we applied multiple strategies: comprehensive unit testing to validate the functionality of the generated code samples, manual filtering to remove any that did not meet our criteria for functional correctness and implementation conciseness, and selecting tasks from the popular benchmark CoderEval, ensuring their high quality and relevance.

LLM Utilization. Another potential threat is the utilization (e.g., source, parameter settings) of the LLMs used in our study. We carefully used the official release versions of each model to avoid any potential issues with unofficial or modified versions, followed the guidelines provided by the model developers to ensure proper implementation and usage, and conducted repeated tests to verify the performance and consistency of the models' outputs. To ensure a fair comparison, we used the same prompt structure and generation parameters for each model, standardizing the experimental setup across different models.

Taxonomy Reliability and Completeness. The reliability and completeness of the inconsistency types identified pose another potential threat. We employed the open coding methodology to systematically identify and categorize inconsistency types, adhered to established open coding practices to ensure thoroughness and accuracy, and ensured that our taxonomy was stable by iteratively refining the inconsistency types until no new categories emerged. We involved multiple annotators to score these metrics, and they discussed their ratings to reach a consensus, reducing individual biases and ensuring more objective assessments. To further bolster the credibility of our findings, we have made all our data publicly available, allowing others to verify our results and methodology, thus enhancing the robustness of our conclusions.

6 Conclusion

Many studies have focused on improving the functional correctness of LLM-based code generation. However, the code style of LLMs—an important aspect of code quality that extends beyond functional correctness—remains under-explored. To fill this gap, this paper makes the first attempt to investigate the coding style differences between LLMs and human developers through an empirical study. Specifically, we compare the code generation results of five mainstream LLMs with ground truth on the CoderEval benchmark.

We present a comprehensive taxonomy of coding style inconsistencies between LLMs and human developers, identifying 24 inconsistency types across five dimensions. We adopt the definition of code style from previous work, but our taxonomy offers a more detailed classification of code style. Our analysis reveals clear coding style differences between the studied LLMs and human developers, particularly in statements/expressions and formatting, while showing similar coding styles among the studied LLMs. We further discuss potential causes of these style inconsistencies and explore ways to improve coding style discrepancies through prompt engineering, providing a foundation for future research in this area.

7 Data Availability

All the data used in this study is provided in the replication package [1].

Acknowledgments

The work is supported by CCF-Huawei Populus Grove Fund CCFHuaweiSE202403, the Natural Science Foundation of Guangdong Province under Grant 2024A1515010255, and National Natural Science Foundation of China under Grant No. 62402113.

References

[1] 2024. *Replication Package*. https://github.com/DeepSoftwareAnalytics/Coding-Style-Empirical

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[5] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. 2019. Python coding style compliance on stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 210–214.

[6] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, et al. 2023. Codeplan: Repository-level coding using llms and planning. *arXiv preprint arXiv:2309.12499* (2023).

[7] R E Berry and B AE Meekings. 1985. A style analysis of C programs. *Commun. ACM* 28, 1 (1985), 80–88.

[8] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).

[9] Binger Chen and Ziawasch Abedjan. 2023. DUETCS: Code Style Transfer through Generation and Retrieval. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2362–2373.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Penglong Chen, Zhen Li, Yu Wen, and Lili Liu. 2022. Generating adversarial source programs using important tokens-based structural transformations. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 173–182.

[12] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232* (2023).

[13] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[14] Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. 2023. Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. *arXiv preprint arXiv:2308.04451* (2023).

[15] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *arXiv preprint arXiv:2304.07590* (2023).

[16] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).

[17] Jing Gong, Yanghui Wu, Linxi Liang, Zibin Zheng, and Yanlin Wang. 2024. CoSQA+: Enhancing Code Search Dataset with Matching Code. *arXiv preprint arXiv:2406.11589* (2024).

[18] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[19] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[20] Lianghong Guo, Yanlin Wang, Ensheng Shi, Wanjun Zhong, Hongyu Zhang, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. When to stop? towards efficient code generation in llms with excess token prevention. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1073–1085.

[21] Rajarshi Haldar and Julia Hockenmaier. 2024. Analyzing the performance of large language models on code summarization. *arXiv preprint arXiv:2404.08018* (2024).

[22] Fan Hu, Yanlin Wang, Lun Du, Hongyu Zhang, Dongmei Zhang, and Xirong Li. 2024. Tackling Long Code Search with Splitting, Encoding, and Aggregating. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. 15500–15510.

[23] Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing Large Language Models in Coding Through Multi-Perspective Self-Consistency. *arXiv preprint arXiv:2309.17272* (2023).

[24] Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. 2024. KareCoder: A New Knowledge-Enriched Code Generation System. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 270–271.

[25] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. 2023. LLM-Assisted Code Cleaning For Training Accurate Code Generators. *arXiv preprint arXiv:2311.14904* (2023).

[26] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su. 2021. Exploring dynamic selection of branch expansion orders for code generation. *arXiv preprint arXiv:2106.00261* (2021).

[27] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).

[28] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689* (2023).

[29] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23, 2009 (2009).

[30] Mizuki Kondo, Daisuke Kawahara, and Toshiyuki Kurabayashi. 2024. Improving Repository-level Code Search with Text Conversion. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*. 130–137.

[31] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328.

[32] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured Chain-of-Thought Prompting for Code Generation. *arXiv preprint arXiv:2305.06599* (2023).

[33] Jia Li, Ge Li, Chongyang Tao, Huangzhao Zhang, Fang Liu, and Zhi Jin. 2023. Large Language Model-Aware In-Context Learning for Code Generation. *arXiv preprint arXiv:2310.09748* (2023).

[34] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2404.00599* (2024).

[35] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv e-prints* (2024), arXiv–2405.

[36] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144* (2023).

[37] Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. 2024. Do Machines and Humans Focus on Similar Code? Exploring Explainability of Large Language Models in Code Summarization. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 47–51.

[38] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. Towards Enhancing In-Context Learning for Code Generation. *arXiv preprint arXiv:2303.17780* (2023).

[39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[40] Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023. Think Outside the Code: Brainstorming Boosts Large Language Models in Code Generation. *arXiv preprint arXiv:2305.10679* (2023).

[41] Yifan Li, Ensheng Shi, Dewu Zheng, Kefeng Duan, Jiachi Chen, and Yanlin Wang. 2024. RepoMinCoder: Improving Repository-Level Code Generation Based on Information Loss Screening. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*. 229–238.

[42] Zehan Li, Jianfei Zhang, Chuantao Yin, Yuanxin Ouyang, and Wenge Rong. 2024. ProCQA: A Large-scale Community-based Programming Question Answering Dataset for Code Search. *arXiv preprint arXiv:2403.16702* (2024).

[43] Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024. STALL+: Boosting LLM-based Repository-level Code Completion with Static Analysis. *arXiv preprint arXiv:2406.10018* (2024).

[44] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[45] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 434–445. doi:10.1109/ASE56229.2023.00159

[46] Cristina V Lopes, Vanessa I Klotzman, Iris Ma, and Iftekar Ahmed. 2024. Commit Messages in the Age of Large Language Models. *arXiv preprint arXiv:2401.17622* (2024).

[47] Abhinav Reddy Mandli, Saurabhsingh Rajput, and Tushar Sharma. 2024. COMET: Generating Commit Messages using Delta Graph Context Representation. *arXiv preprint arXiv:2402.01841* (2024).

[48] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 468–478.

[49] Qing Mi, Jacky Keung, and Yang Yu. 2016. Measuring the stylistic inconsistency in software projects using hierarchical agglomerative clustering. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. 1–10.

[50] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. ClarifyGPT: Empowering LLM-based Code Generation with Intention Clarification. *arXiv preprint arXiv:2310.10996* (2023).

[51] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.

[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[53] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. 2023. Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Models. *arXiv preprint arXiv:2312.06227* (2023).
[54] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. *arXiv preprint arXiv:2306.09896* (2023).
[55] Paul W Oman and Curtis R Cook. 1990. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*. 244–250.
[56] OpenAI. 2021. OpenAI Code. https://openai.com/blog/openai-code.
[57] Terence Parr and Jurgen Vinju. 2016. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 137–151.
[58] Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. RepoHyper: Better Context Retrieval Is All You Need for Repository-Level Code Completion. *arXiv preprint arXiv:2403.06095* (2024).
[59] F Rohlf et al. 1981. Biometry the principles and practice of statistics in biological research. (1981).
[60] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[61] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).
[62] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond. *arXiv preprint arXiv:2304.05216* (2023).
[63] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, FA Rifat, and V Carvalho Lopes. 2023. Exploring the effectiveness of large language models in generating unit tests. *arXiv preprint arXiv:2305.00418* (2023).
[64] KR Srinath. 2017. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology* 4, 12 (2017), 354–357.
[65] Chia-Yi Su and Collin McMillan. 2024. Distilled GPT for source code summarization. *Automated Software Engineering* 31, 1 (2024), 22.
[66] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865* (2023).
[67] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, and Li Li. 2023. Don't Complete It! Preventing Unhelpful Code Completion for Productive and Sustainable Neural Code Completion Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 324–325.
[68] Karl Tamberg and Hayretdin Bahsi. 2024. Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. arXiv:2405.15614 [cs.CR] https://arxiv.org/abs/2405.15614
[69] Wei Tao, Yucheng Zhou, Yanlin Wang, Hongyu Zhang, Haofen Wang, and Wenqiang Zhang. 2024. KADEL: Knowledge-Aware Denoising Learning for Commit Message Generation. *ACM Transactions on Software Engineering and Methodology* (2024).
[70] Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. 2024. Structcoder: Structure-aware transformer for code generation. *ACM Transactions on Knowledge Discovery from Data* 18, 3 (2024), 1–20.
[71] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving llm code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632* (2024).
[72] Yanlin Wang, Lianghong Guo, Ensheng Shi, Wenqing Chen, Jiachi Chen, Wanjun Zhong, Menghan Wang, Hui Li, Hongyu Zhang, Ziyu Lyu, et al. 2023. You Augment Me: Exploring ChatGPT-based Data Augmentation for Semantic Code Search. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 14–25.
[73] Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. 2024. SparseCoder: Identifier-Aware Sparse Transformer for File-Level Code Summarization. *arXiv preprint arXiv:2401.14727* (2024).
[74] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* (2024).
[75] Ziliang Wang, Ge Li, Jia Li, Yingfei Xiong, and Zhi Jin. 2024. M2CVD: Multi-Model Collaboration for Code Vulnerability Detection. *arXiv preprint arXiv:2406.05940* (2024).
[76] Zejun Wang, Jia Li, Ge Li, and Zhi Jin. 2023. ChatCoder: Chat-based Refine Requirement Improves LLMs' Code Generation. *arXiv preprint arXiv:2311.00272* (2023).

[77] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).
[78] Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Xiaofei Ma, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, et al. 2023. Exploring continual learning for code generation models. *arXiv preprint arXiv:2307.02435* (2023).
[79] Aidan Z. H. Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. 2024. Security Vulnerability Detection with Multitask Self-Instructed Fine-Tuning of Large Language Models. arXiv:2406.05892 [cs.CR] https://arxiv.org/abs/2406.05892
[80] Rafed Muhammad Yasir and Dr Ahmedul Kabir. 2022. Exploring the Impact of Code Style in Identifying Good Programmers. *arXiv preprint arXiv:2206.10891* (2022).
[81] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[82] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).
[83] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).
[84] Imam Nur Bani Yusuf and Lingxiao Jiang. 2024. Your Instructions Are Not Always Helpful: Assessing the Efficacy of Instruction Fine-tuning for Software Vulnerability Detection. arXiv:2401.07466 [cs.SE] https://arxiv.org/abs/2401.07466
[85] Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2023. Private-library-oriented code generation with large language models. *arXiv preprint arXiv:2307.15370* (2023).
[86] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7443–7464.
[87] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. arXiv:2303.12570 [cs.CL]
[88] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.
[89] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. *arXiv preprint arXiv:2305.04087* (2023).
[90] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. In *The Eleventh International Conference on Learning Representations*.
[91] Yuxia Zhang, Zhiqing Qiu, Klaas-Jan Stol, Wenhui Zhu, Jiaxin Zhu, Yingchen Tian, and Hui Liu. 2024. Automatic commit message generation: A critical review and directions for future work. *IEEE Transactions on Software Engineering* (2024).
[92] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. *arXiv preprint arXiv:2308.11396* (2023).
[93] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).
[94] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. *arXiv preprint arXiv:2311.10372* (2023).
[95] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.

Received 2024-09-13; accepted 2025-01-14