`code_transformed`: The Influence of Large Language Models on Code

Anonymous EMNLP submission

Abstract

Coding remains one of the most fundamental modes of interaction between humans and machines. With the rapid advancement of *Large Language Models* (LLMs), code generation capabilities have begun to significantly reshape programming practices. This development prompts a central question: *Have LLMs transformed code style, and how can such transformation be characterized?* In this paper, we present a pioneering study that investigates the impact of LLMs on code style from the perspectives of naming conventions, complexity and maintainability, and structural similarity. By analyzing code from over 19,000 GitHub repositories linked to arXiv papers published between 2020 and 2025, we identify measurable trends in the evolution of coding style that align with characteristics of LLM-generated code. For instance, the proportion of snake_case variable names in Python code increased from 47% in Q1 2023 to 51% in Q1 2025. Furthermore, we extend our analysis to examine whether LLM-generated content influences their subsequent code generation behavior. Our experimental results provide the first large-scale empirical evidence that LLMs affect real-world programming style.

1 Introduction

Coding serves as one of the most fundamental interaction pathways between humans and machines. Recently, coding practices have undergone a significant transformation with the emergence of tools like Copilot and Cursor, powered by *Large Language Models* (LLMs) (Liang et al., 2024a; Daigle and Staff, 2024; Peslak and Kovalchick, 2024; Li et al., 2025; Chen et al., 2021), including state-of-the-art models such as GPT-4o (Hurst et al., 2024), DeepSeek-Coder (Guo et al., 2024), and Gemini (Team et al., 2023).

Despite significant advancements, the widespread adoption of LLMs has raised concerns about code integrity (Wadhwa et al., 2024), potential copyright infringement (Wan et al., 2024), and broader ethical or legal implications (Xu et al., 2024b). These issues have motivated efforts to trace and attribute the influence of LLM-assisted programming. Moreover, this phenomenon suggests the intriguing possibility that software itself may be evolving under the influence of LLMs—not only through their direct use in code generation but also via indirect exposure to LLM-generated content. If substantiated, this would position LLMs not merely as tools for writing code, but as influential agents shaping human coding practices and stylistic norms.

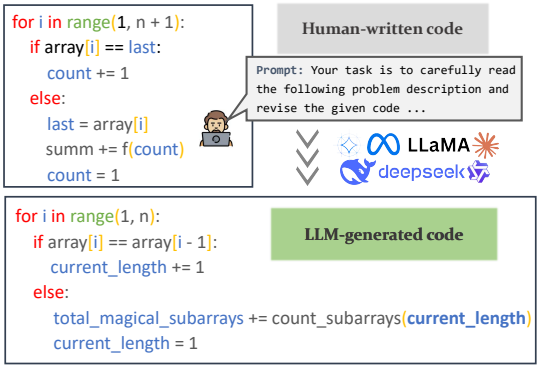


Figure 1: An example of LLMs' preference for longer variable names and *snake_case* naming patterns.

Figure 1 presents a motivating example illustrating that content generated by LLMs differs from human-written code, supporting prior observations that LLMs exhibit distinct coding styles (Wang et al., 2024). As shown in the figure, LLMs tend to replace short variable names with longer, more descriptive names following the *snake_case* convention (i.e., `current_length`, and `total_magical_subarrays`). This aligns with findings from recent studies, which highlight LLMs' preferences in naming consistency, code structure, and overall readability (Park et al., 2025).

Building on prior research that explores the influence of LLMs in text and speech domains (Liang

et al., 2024b; Geng et al., 2024), this paper turns to a research question in the programming context: *Have LLMs transformed code style, and how can such transformation be characterized?*

To address this question, we conduct a pioneering study to investigate the influence of LLMs on code, from the views of naming patterns, complexity and maintainability, and code similarity. Furthermore, we extend our analysis to examine whether the generated content affects the subsequent code generation capabilities of LLMs.

From the view of naming patterns, we first categorize variable, function, and file names into several distinct formats: *single letter*, *lower-case*, *UPPERCASE*, *camelCase*, *snake_case*, *PascalCase*, and *endsWithDigits*. By analyzing the names of variables and functions in GitHub repository code, we observe a clear increase in the usage of LLM-preferred naming styles.

From the perspective of code complexity and maintainability, we simulate code generated and rewritten by LLMs, extract subsets for analysis, and compare them with human-written code from GitHub. Our results indicate that LLM-rewritten code tends to be more concise under certain metrics—notably, cyclomatic complexity in Python. However, this improvement is less pronounced in stylistic aspects such as naming conventions. Additionally, no clear trend is observed in the GitHub code, suggesting that LLMs may not differ substantially from human developers in these dimensions.

From the perspective of code similarity, the rewritten code exhibits relatively high similarity to the original, especially when compared to code generated directly by LLMs. This observation further highlights that different usage scenarios—such as code rewriting versus direct generation—can yield distinct outcomes. These findings may offer valuable insights for future efforts to detect and distinguish how LLMs are utilized, whether for assisted programming.

Based on the above findings, we further explore whether the generated content influences the subsequent code generation abilities of LLMs. By analyzing the models' reasoning process, we find that their outputs do not always align with the expected algorithmic approaches for the given problems.

We believe the findings in our work will enhance knowledge about LLMs' programming abilities and coding styles, providing novel insights for assessing and monitoring their broader impacts. To facilitate further study, the experimental dataset and source code will be made available.

2 Background

2.1 Position of Our Work

Comparisons between code generated by LLMs and that written by humans can be conducted from multiple perspectives. Prior research has investigated various methods to distinguish between the two, such as leveraging perplexity scores (Xu and Sheng, 2024) and manually designed features (Bulla et al., 2024; Park et al., 2025). However, rather than focusing on differentiating LLM-generated code from human-written code, our work considers a more realistic and increasingly common scenario: LLM-assisted code authoring, where human developers and language models collaboratively produce code.

In this paper, we investigate coding style from observable perspectives, such as naming patterns and Cyclomatic complexity (McCabe, 1976). With regard to code-level metrics, Halstead complexity metrics provide a quantitative assessment of code complexity based on the use of operands and operators (Hariprasad et al., 2017). Graylin et al. (2009) investigate the relationship between Cyclomatic Complexity and lines of code. The Maintainability Index is a calculation used to review the level of maintenance of the software (Kencana et al., 2020). Additionally, we also consider comparing the similarity between human-written code and code that has been rewritten or generated by LLMs using cosine similarity and Jaccard similarity. In order to better grasp how LLMs generate code, we also carefully analyze the reasoning chains to see if they think about the algorithms that the problems were designed to elicit.

2.2 Code Style Measurements

Naming Patterns. Studies have pointed out that the naming in code generated by LLMs has its own characteristics (Park et al., 2025). Therefore, we categorize variable, function, and file names into several distinct formats (*e.g. snake_case*). The length of the names has also been considered.

Cyclomatic Complexity. Cyclomatic complexity is a metric used to measure the number of linearly independent paths in the code. Some researchers use this method to analyze the code generated by LLMs (Dou et al., 2024). Given the control-flow graph (CFG) of a code snippet, let E denote the number of edges, n the number of nodes, and P

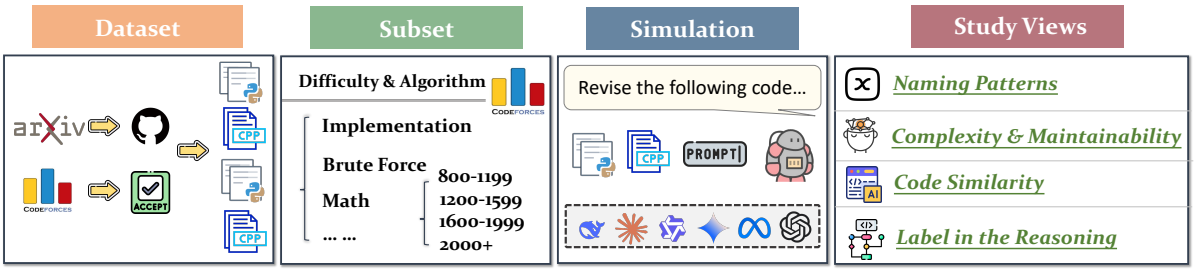


Figure 2: The process of our experiments.

the number of connected components. The cyclomatic complexity is calculated by $G = E - N + 2P$. For a single connected component ($P = 1$), it simplifies to the number of decision points plus one. Each occurrence of `if`, `for`, `while`, `case`, etc., is counted as one decision point.

Code Similarity. Let A and B be the words of the code segment, and let $\vec{v}_A$ and $\vec{v}_B$ be their corresponding vector representations. Then we can use the following cosine similarity to compare the similarity of the code, defined as follows:

$$\text{sim}_{\text{cosine}}(A, B) = \frac{\vec{v}_A \cdot \vec{v}_B}{\|\vec{v}_A\| \cdot \|\vec{v}_B\|} \quad (1)$$

Similarly for Jaccard similarity:

$$\text{sim}_{\text{J}}(A, B) = \begin{cases} 1, \text{ if } A = \emptyset \text{ and } B = \emptyset \\ 0, \text{ if } A = \emptyset \text{ or } B = \emptyset \\ \frac{|A \cap B|}{|A \cup B|}, \text{ otherwise} \end{cases} \quad (2)$$

Label Similarity. In order to further refine our analysis, we analyzed the matching of reasoning and labels for each question separately. The specific approach is as follows: take all collected tags as the full set, if the tag corresponding to the question appears in the reasoning, it is considered a match, and if a tag in the full set but not for this question appears, it is considered an error.

Let T denote the set of all labels. For each question q, let $A_q \subseteq T$ be the set of true labels in the question description, and let $R_q \subseteq T$ be the set of labels in reasoning process. Then we define the match and error metrics as follows:

$$\text{match}(q) = \mathbf{1}\left(A_q \cap R_q \neq \varnothing\right), \quad (3)$$

$$\text{error}(q) = \mathbf{1}\left((T \setminus A_q) \cap R_q \neq \varnothing\right), \quad (4)$$

where $\mathbf{1}(\cdot)$ is the indicator function: 1 if the condition is met, 0 otherwise.

3 Study Design

Figure 2 illustrates the process of our experiment. We begin by collecting human-written code from GitHub and Codeforces, and then generate code using three LLMs under different prompting strategies. By comparing the differences in code metrics between human and LLM-generated solutions, and analyzing the temporal trends of these metrics on GitHub, we investigate the relationship between the two. Furthermore, to broaden the scope of our study, we select a subset of problems and involve a larger set of models to explore stylistic differences across LLM-generated code.

3.1 Dataset

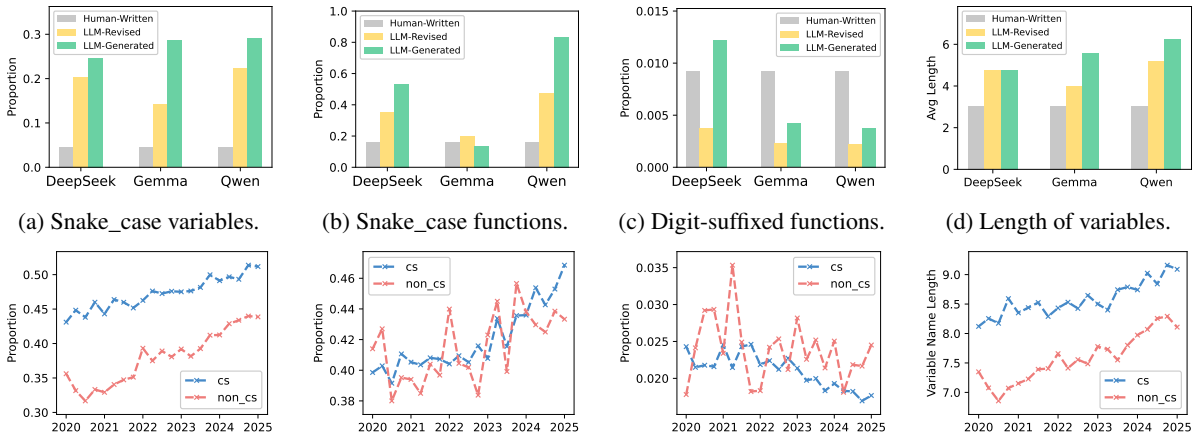
Human-Written Code. We utilize `Code4Bench`, a multidimensional benchmark based on Codeforces data (Majd et al., 2019). This dataset contains user submissions on Codeforces before 2020, Which were barely impacted by LLMs.

GitHub Coding Data. We collect GitHub repository links by matching them from the abstract and comment fields in the arXiv dataset[1]. Our dataset contains a total of 19,898 GitHub repositories and 926,935 source code files, corresponding to arXiv papers from the first quarter of 2020 to the first quarter of 2025.

Each repository in the dataset is labeled with two attributes: the programming language, which can be either Python or C/C++, and the scientific domain, indicating whether the associated arXiv paper belongs to the field of computer science (cs) or a non-computer science category (non-cs). The number of repositories and files of each language per quarter based is shown in Table 3.

There are some issues in our dataset collection process. In the arXiv dataset, a single GitHub link may appear multiple times, likely because the same repository was used for multiple paper submissions. For such repositories, if the publication dates of all associated papers fall within a two-quarter range, we retain the link and assign it the most recent publication date as its timestamp. Otherwise, we

[1] https://www.kaggle.com/datasets/Cornell-University/arxiv/data



(a) Snake_case variables. (b) Snake_case functions. (c) Digit-suffixed functions. (d) Length of variables.

(e) Snake_case variables. (f) Snake_case functions. (g) Digit-suffixed functions. (h) Length of variables.

Figure 3: The four figures in the first row present simulation results derived from Codeforces human-written code, either revised by LLMs or directly generated by LLMs based on problem descriptions. The four figures in the second row illustrate the trends over time in GitHub repositories for Python variable names using *snake_case*, digit-suffixed function names, and the length of variable names.

discard the repository. Repositories that lack target language files or are excessively large are also excluded from our analysis.

Problem Subset. To reduce computational costs while maintaining representativeness, we select 200 questions from `Code4Bench`, spanning a range of difficulty levels and algorithm types and categorize them into four groups based on their difficulty rating: 800–1199, 1200–1599, 1600–1999, and 2000+. The first valid tag is utilized to determine each problem's primary algorithm.

We then filter problems that are annotated with one of the following ten target algorithms: *implementation, brute force, constructive algorithms, greedy, binary search, math, dp, data structures, combinatorics, and dfs and similar*. From each difficulty group, we randomly sample 50 problems. The number of problems per algorithm is proportional to its distribution within that group. Detailed information is provided in table 4.

3.2 Studied LLMs

We select a diverse set of models to cover a range of architectures and parameter scales. The *Qwen3* series (4B, 8B, 14B, 32B) and *Qwen2.5-Coder-32B-Instruct* are chosen for exploring the impact of model sizes on code genertion (Yang et al., 2025a; Hui et al., 2024). The DeepSeek family, including *DeepSeek-V3*, *DeepSeek-R1*, and *DeepSeek-R1-Distill-Qwen-32B*, was included to evaluate the reasoning performance. (DeepSeek-AI, 2024, 2025). We also incorporate leading general-purpose LLMs, *GPT-4o-mini* (Hurst et al., 2024) and *Claude-3.7-Sonnet* (Anthropic, 2024), as well as *Gemma-3-27B* (Team et al., 2025) and *Llama-3.3-Nemotron-Super-49B-V1* (Bercovich et al., 2025), to ensure broad coverage of both closed- and open-source training paradigms.

3.3 Simulations

We implement two strategies for code generation using LLMs:

Direct Generation. LLMs are provided only with the problem description and asked to generate a solution from scratch.

Reference-Guided Generation. In addition to the problem description, the model is also given a reference solution (i.e., a user-submitted, passed code). The model is instructed to analyze this code and revise it when generating its solution.

We applied both strategies to three LLMs: *Deepseek-R1-Distill-Qwen-32B*, *Qwen3-32B*, and *Gemma-3-27B*. The prompt for the two strategies are shown in Figures 7 and 8.

3.4 Evaluation

In addition to examining the similarities and differences between human-written code and LLM-generated code, we also aim to understand how different LLMs compare with each other, and whether some models produce code that more closely resembles human style. Therefore, we conduct a larger-scale evaluation across a broader set of models. Based on our benchmark, we expanded our analysis to include all models in Table 3.2 except

DeepSeek-R1-Distill-Qwen-32B. Here, we do not record the reasoning process, nor do we require the model to rewrite human-written code. We only prompt the models to generate code for each problem, repeating the process 32 times. The prompt is shown in Figure 9.

4 View I: Naming Patterns

4.1 Settings

We categorize variable, function, and file names into several distinct formats: *single letter*, *lowercase*, *UPPERCASE*, *camelCase*, *snake_case*, *PascalCase*, and *endsWithDigits*. Any name that does not match these specific patterns is grouped into the *Other* category. The length of the names is also considered as an additional metric.

To extract names from source code, we apply different strategies depending on the programming language. For Python code, we use the `ast` module to statically parse the abstract syntax tree and extract function and variable names. For C/C++ code, we use regular expressions to identify name patterns directly from the source text. All extracted names are then matched against predefined regular expressions to classify them into the aforementioned naming formats.

To prevent large repositories from dominating the overall distribution, we normalize at the repository level: we first compute naming pattern distributions for each file, average them within each repository, and then average across repositories to obtain overall statistics.

4.2 Results

Naming Patterns in LLM-generated Code. LLMs have slight deviations from general human naming conventions when it comes to variables and functions, for instance, Figures 3a and 3b illustrate that all three evaluated LLMs tend to use *snake_case* in names compared to human-written code. Figure 3d shows that LLMs tend to use longer variable names. But there isn't always a clear demarcation, for example, *digit-suffixed* naming pattern plotted in Figure 3c.

Trends in GitHub Repositories. Figures 3e and 3f show the adoption of *snake_case* names steadily rises in both CS and non-CS projects, which is consistent with the stylistic differences observed between human-written code and LLM-generated code. Similarly, Figure 3h presents the growth in the length of variable names in GitHub code.

Differences between Programming Languages. Unlike Python, fewer naming patterns show clear temporal trends in C/C++ repositories. However, there are still notable cases: the use of *snake_case* in both variable and function names shows an upward trend as shown in Figures 13 and 14, while the use of lowercase names in variables declines over time. Both of them align with the stylistic tendencies observed in LLM-generated code.

Influence of Disciplines. For non-CS repositories, naming patterns sometimes exhibit greater fluctuation compared to the clearer trends observed in CS repositories. The trend of Python function names ending with digits, as shown in Figure 3c, is a good example. It shows a steady decline in CS repositories, while exhibiting greater variability in non-CS projects. These results suggest that Python Code and CS Repositories Appear to be More Susceptible to LLM Influence than C++ Code and Non-CS Repositories.

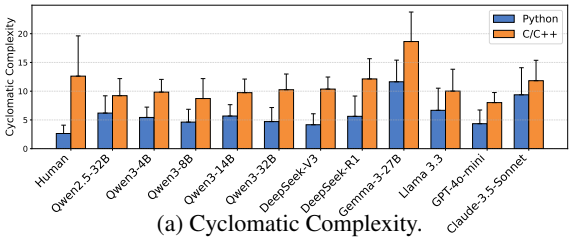
Other Evidences. There are more examples illustrating how the evolution of human code naming patterns increasingly aligns with the stylistic tendencies of LLM-generated code. Figure 10 show that all three LLMs tend to avoid *single-letter* and *digit-suffixed* variable names, and these patterns are also steadily declining in both CS and non-CS repositories. Similar trends are observed in function names. At least two of the models favor the use of *snake_case* and show a clear tendency to avoid *lowercase* naming styles. GitHub repositories reflect this shift as well, with *snake_case* usage showing a consistent upward trend, while the use of *lowercase* names gradually decreases. These parallel developments suggest a potential correlation between human and machine-generated coding styles. Full results and analysis are presented in Section C in the appendix.

> ***Finding 1:*** The coding style of human-written code may be influenced by LLMs: they may not only mirror existing norms but also subtly reshape them, gradually pushing human developers toward greater stylistic alignment with LLM-preferred conventions.

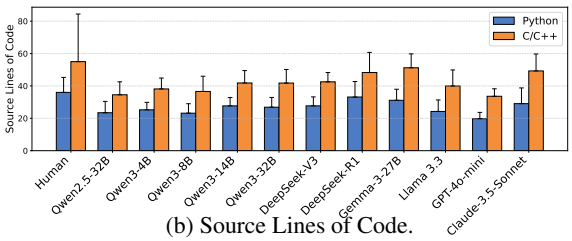
5 View II: Complexity and Maintainability

5.1 Settings

In order to explore the quality difference between code written by humans and LLM, we adopt two

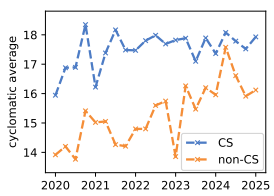


(a) Cyclomatic Complexity.

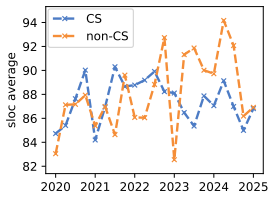


(b) Source Lines of Code.

Figure 4: Results on our subset based on *cyclomatic complexity* and *source lines of code* metrics. The names *Llama-3.3-Nemotron-Super-49B-V1* and *Qwen2.5-Coder-32B-Instruct* are abbreviated as Llama-3.3 and Qwen2.5-32B.



(a) Cyclomatic Complexity.



(b) Source Lines of Code.

Figure 5: Temporal trends of *cyclomatic complexity* and *source lines of code* in GitHub repositories.

text-based scoring methods that assess code maintainability from multiple dimensions, including information volume, control flow complexity, code structure, and adequacy of comments. We first calculate the mean and standard deviation for each problem across 32 generated outputs per model, using metrics defined in Section 2 and Appendix A. Subsequently, we average these per-problem statistics across all problems to obtain final results for each model. These final values reflect each model's overall performance on the generation task, as well as the stability of its output for individual problems.

5.2 Results

LLM-Generated *v.s* LLM-Revised Tables 9 and 10 show the results containing reference-guided generation . From the tables, we can see that for the same model, in Python and C++, the Halstead volume and effort of the solution obtained using the user AC code as the reference (REF) are often higher than those of direct ANS. For example, the volume (192.36) and effort (1367.70) of Deepseek-REF-Python are higher than those of Deepseek-ANS-Python (volume: 101.22, effort: 631.26), indicating that the **LLM-modified code is not as concise as the original output**; while in C++, the MI of REF does not change much compared with ANS, both in the range of 38–42.

LLM *v.s* Human Tables 11 and 12 show the results of large-scale model evaluation. Figure 4a shows the the cyclomatic complexity of the code written by human and different models. Compared to code generated by various LLMs, the cyclomatic complexity of human-written C/C++ code ranks second highest. And the standard deviation of human-written code remains consistently high, indicating greater variability. Figure 4b shows the source lines of code of the code written by human and different models, from it we can also find human-written code also tends to have longer source lines of code (both Python and C/C++), indicating that LLM-generated code is more concise.

Compared with the GitHub Dataset. Figure 5 shows the trend of cyclomatic index and sloc index in our github dataset. We found that there is no obvious trend in the cyclomatic index and sloc index over time, which may indicate that LLM is still limited in its role in improving code maintainability in complex scenarios such as GitHub repositories.

In addition, from Figure 5a, we found that the cyclomatic complexity of the CS category is always significantly higher than that of the non-CS category. In Figure 5b, the indicators of the two categories are not much different.

> ***Finding 2:*** LLM's code writing has lower complexity and higher maintainability than humans in the scenario of IO algorithm problems. At the same time, the output is stable, and its rewritten code indicators are inferior to direct generation.

6 View III: Code Similarity

6.1 Settings

To quantify how closely LLM outputs mirror human style and how much they benefit from seeing a human solution, we compare three versions of each problem's code: the original human-authored solution (**AC**), the LLM's output given only the problem description (**ANS**), and the LLM's output when additionally conditioned on the human solution (**REF**). The methods for generating and rewriting code are defined in Section 3.3. We mea-

sure pairwise cosine and Jaccard similarities among AC, ANS, and REF (Park et al., 2025).

6.2 Results

Tables 1 and 5 show the cosine similarity and jaccard similarity between AC, ANS and REF. We can see that the overall trends of cosine similarity and Jaccard similarity are consistent. Among the three pairwise comparisons, AC vs REF yields the highest similarity, indicating that **LLMs are capable of imitating a given human-written solution when it is provided**. In contrast, AC vs ANS exhibits the lowest similarity and remains relatively low overall, suggesting that in the context of IO algorithm programming tasks, LLM-generated code, when produced without reference, **differs substantially in style from human-written code**. Additionally, similarity scores vary across models. For example, all three similarity values for *Gemma-27B* are noticeably higher than those of other models, indicating not only that its generated code most closely resembles human code, but also that it **demonstrates the strongest capacity for learning and imitation** when reference code is available.

> ***Finding 3:*** LLMs can effectively mimic human coding style when given reference code, but without such guidance, their generated solutions diverge significantly from human-written code—especially in IO algorithm tasks.

7 Labels in the Reasoning Process

7.1 Settings

In addition to the final generated code, the reasoning process of the model can also reveal how LLMs understand and solve coding problems. First, each problem on codeforces has a corresponding algorithm label, and a problem may have multiple algorithm labels corresponding to different solutions. Since the original code4bench dataset does not contain this part, we extend it by adding new information. Then, for the model's reasoning process, we match whether these labels appear in it. After counting each problem, we average all the match rates and error rates, count the rate of match and error, and calculate the match rate and error rate of questions of various difficulty levels separately. The definition of "match and error" and methods for calculating are defined in Section 2.1.

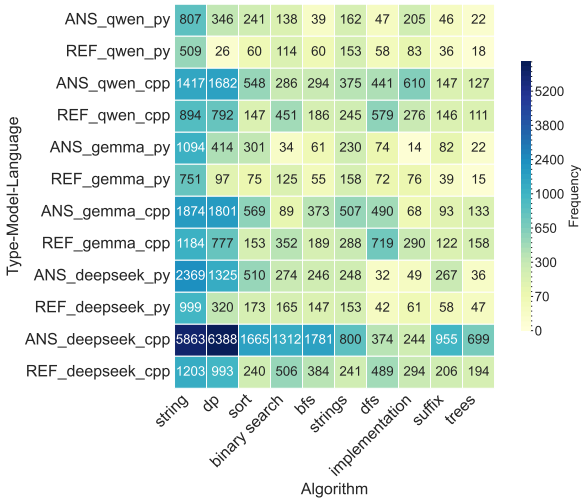


Figure 6: Frequency comparison of top 10 algorithms on various models (ANS/REF, Python/C/C++).

7.2 Results

Tags Frequencies. Table 6 shows the tag frequency of the collected questions, and Table 7 shows the tag frequency results of the model output reasoning. Figure 6 shows the frequency of the 10 most common algorithm labels in each field for each model. We can see that most of the word frequencies ANS is higher than REF, indicating that **the reference code-based model tends to analyze without relying on algorithms**. Additionally, C/C++'s word frequency is always higher than Python, indicating that the **model is accustomed to analyzing from an algorithmic perspective when implementing C/C++ code**. Furthermore, deepseek's output frequency in these ten tags is significantly higher than qwen and gemma, and string and dp even exceed 5000 and 6000 respectively, indicating that **its algorithm-based thinking performs best in IO scenarios**.

Match and Error Rate. Table 2 Shows the match and error rate between the label and the model output reason. Table 8 shows specific results at different levels of difficulty. From the result, we can see that the error rate generally exceeds the matching rate, suggesting that **LLMs tend to explore more incorrect approaches**, likely due to their reliance on a limited set of mainstream algorithms. Additionaly, the matching performance varies across models: Qwen achieves a higher match rate when generating from scratch (ANS) than with reference (REF), whereas Gemma and Deepseek show the opposite trend. This indicates that **some models prioritize extracting high-level ideas from reference code, while others focus**

Model	C/C++			Python		
	AC vs ANS	AC vs REF	ANS vs REF	AC vs ANS	AC vs REF	ANS vs REF
Qwen-32B	0.2752	0.6140	0.3626	0.2038	0.5448	0.2849
Gemma-27B	**0.2958**	**0.7431**	0.3826	0.2541	0.7251	0.3106
DeepSeek-32B	0.1789	0.5603	0.2366	0.1758	0.4092	0.2370

Table 1: Cosine similarity between the human-written code (AC), the initial LLM-generated code (ANS), and the LLM-rewritten code (REF) based on human-written code.

Model	ANS (Python)		REF (Python)		ANS (C/C++)		REF (C/C++)	
	Match	Error	Match	Error	Match	Error	Match	Error
Qwen-32B	17.40%	25.15%	12.06%	16.67%	27.49%	35.84%	26.15%	24.71%
Gemma-27B	10.96%	16.67%	11.33%	13.52%	17.11%	20.79%	25.18%	23.34%
DeepSeek-32B	10.75%	15.57%	12.65%	15.28%	19.61%	27.95%	27.76%	25.43%

Table 2: Match and error rates between the model's predicted reasoning (ANS or REF) and the ground-truth algorithm labels, for both Python and C/C++ code.

more on implementing specific details. Furthermore, both the matching and error rates are higher for C++ than for Python, which may reflect **language-specific design choices in the models-favoring algorithmic reasoning in C++ and practical implementation in Python**.

Finding 4: LLMs have low algorithm analysis capabilities, are more inclined to approach C/C++ code from an algorithmic perspective, and harder problems may better activate their algorithmic reasoning capabilities.

8 Related Work

LLMs for Code Generation. Code generation has been seen rapid progress in recent years. Before ChatGPT arrived, Transformer-based models for code generation had already been developed, such as CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), Codex (Chen et al., 2021). Research and discussions on the use of LLMs for code generation are ongoing (Liu et al., 2023; Jiang et al., 2024). Meanwhile, new LLM models always consider coding capability as a key evaluation metric (DeepSeek-AI, 2025; Yang et al., 2025b).

LLM-Generated Code Detection. The methods for detecting the code generated by LLMs are diverse, such as feature-based classifiers (Rahman et al., 2024; Demirok and Kutlu, 2024), contrastive learning (Ye et al., 2024; Xu et al., 2024c), Transformer-based encoder classifier (Gurioli et al., 2024). People are also interested in lexical diversity, readability, perplexity, conciseness, and naturalness (Shi et al., 2024; Wang et al., 2024; Xu and Sheng, 2024). Some studies have also pointed out the limitations of these code detection methods (Xu et al., 2024a; Suh et al., 2024)

9 Discussion and Conclusion

Although there have been many papers on the impact of LLMs on text, no other studies have yet examined their effect on code using real-world data. The code generated by LLMs also has its own style, such as in the names of variables and functions. Therefore, we attempted to identify traces of LLMs in the code from GitHub repositories and indeed discovered some evidence. At the same time, due to the diversity of programming languages, usage behaviors, and LLMs, it is difficult to quantitatively estimate how many people are using LLMs to assist in programming.

Other researchers have shown that LLMs can effectively support students in learning coding (Korpimies et al., 2024; Rasnayaka et al., 2024), and the use of LLM-assisted programming tools is likely to increase. The number of questions on Stack Overflow has declined since the emergence of LLMs, which has raised concerns among many[2]. Consequently, there is a strong possibility that human coding style will shift toward that of LLMs in the future. We therefore emphasize the need to consider not only the programming capabilities of LLMs, but also their broader societal implications.

[2] https://blog.pragmaticengineer.com/stack-overflow-is-almost-dead/

Limitations

Although we have conducted multiple experiments to evaluate the changes in code style in the LLM era, our research still has some limitations. First, the one-sided analysis of text style ignores the measurement of the accuracy of the code itself. For example, in the IO question scenario, we use the label overlap rate instead of the pass rate calculation, which is an indirect indicator. The quantitative scoring of the code should not ignore the various parameters of the code runtime. We will improve these issues in future work.

Second, our data set is insufficient. Code4bench is an early user AC code collection, which reflects the poor code style and thinking ability of users in the recent LLM era. The code collection of a single evaluation result cannot reflect the overall style of the entire human code ecology, and the comparison with the LLM-generated code is not complete. We will rebuild our code data set.

In addition, there may be many scenarios for user-generated code and prompt parameters. Our simulation cannot exhaust all user usage situations, and the universality of our research results needs to be further improved.

References

Anthropic. 2024. Anthropic: Introducing claude 3.5 sonnet.

Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, Ido Shahaf, Oren Tropp, Ehud Karpas, Ran Zilberstein, Jiaqi Zeng, Soumye Singhal, Alexander Bukharin, Yian Zhang, Tugrul Konuk, and 114 others. 2025. Llama-nemotron: Efficient reasoning models. *Preprint*, arXiv:2505.00949.

Luana Bulla, Alessandro Midolo, Misael Mongiovì, and Emiliano Tramontana. 2024. Ex-code: A robust and explainable model to detect ai-generated code. *Information*, 15(12):819.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Kyle Daigle and GitHub Staff. 2024. Survey: The ai wave continues to grow on software development teams. *GitHub Blog. Available online: https://github. blog/news-insights/research/survey-ai-wave-grows/# key-survey-findings*.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Basak Demirok and Mucahid Kutlu. 2024. Aigcodeset: A new annotated dataset for ai generated code detection. *arXiv preprint arXiv:2412.16594*.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, and 1 others. 2024. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Mingmeng Geng, Caixi Chen, Yanru Wu, Dongping Chen, Yao Wan, and Pan Zhou. 2024. The impact of large language models in academia: from writing to speaking. *arXiv preprint arXiv:2409.13686*.

Jay Graylin, Randy K SMITH, HALE David, Nicholas A KRAFT, WARD Charles, and 1 others. 2009. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, 2(3):137–143.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Andrea Gurioli, Maurizio Gabbrielli, and Stefano Zacchiroli. 2024. Is this you, llm? recognizing ai-written programs with multilingual code stylometry. *arXiv preprint arXiv:2412.14611*.

T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software complexity analysis using halstead metrics. In *2017 international conference on trends in electronics and informatics (ICEI)*, pages 1109–1113. IEEE.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Gilang Heru Kencana, Akuwan Saleh, Haryadi Amran Darwito, R Rizki Rachmadi, and Elsa Mayang Sari. 2020. Comparison of maintainability index measurement from microsoft codelens and line of code. In *2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI)*, pages 235–239. IEEE.

Kai Korpimies, Antti Laaksonen, and Matti Luukkainen. 2024. Unrestricted use of llms in a software project course: Student perceptions on learning and impact on course performance. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, pages 1–7.

Ruimiao Li, Manli Li, and Weifeng Qiao. 2025. Engineering students' use of large language model tools: An empirical study based on a survey of students from 12 universities. *Education Sciences*, 15(3):280.

Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024a. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13.

Weixin Liang, Zachary Izzo, Yaohui Zhang, Haley Lepp, Hancheng Cao, Xuandong Zhao, Lingjiao Chen, Haotian Ye, Sheng Liu, Zhi Huang, and 1 others. 2024b. Monitoring ai-modified content at scale: A case study on the impact of chatgpt on ai conference peer reviews. *arXiv preprint arXiv:2403.07183*.

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. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Amirabbas Majd, Mojtaba Vahidi-Asl, Alireza Khalilian, Ahmad Baraani-Dastjerdi, and Bahman Zamani. 2019. Code4bench: A multidimensional benchmark of codeforces data for different program analysis techniques. *Journal of Computer Languages*, 53:38–52.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320.

Shinwoo Park, Hyundong Jin, Jeong-won Cha, and Yo-Sub Han. 2025. Detection of llm-paraphrased code and identification of the responsible llm using coding style features. *arXiv preprint arXiv:2502.17749*.

Alan Peslak and Lisa Kovalchick. 2024. Ai for coders: An analysis of the usage of chatgpt and github copilot. *Issues in Information Systems*, 25(4):252–260.

Musfiqur Rahman, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. 2024. Automatic detection of llm-generated code: A case study of claude 3 haiku. *arXiv preprint arXiv:2409.01382*.

Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. 2024. An empirical study on usage and perceptions of llms in a software engineering project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 111–118.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2024. Between lines of code: Unraveling the distinct patterns of machine and human programmers. *arXiv preprint arXiv:2401.06461*.

Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. 2024. An empirical study on automatically detecting ai-generated source code: How far are we? *arXiv preprint arXiv:2411.04299*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, and 1 others. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. Core: Resolving code quality issues using llms. *Proceedings of the ACM on Software Engineering*, 1(FSE):789–811.

Yao Wan, Guanghua Wan, Shijie Zhang, Hongyu Zhang, Pan Zhou, Hai Jin, and Lichao Sun. 2024. Does your neural code completion model use my code? a membership inference approach. *arXiv preprint arXiv:2404.14296*.

Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, and Zibin Zheng. 2024. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *arXiv preprint arXiv:2407.00456*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Jinwei Xu, He Zhang, Yanjin Yang, Zeru Cheng, Jun Lyu, Bohan Liu, Xin Zhou, Lanxin Yang, Alberto Bacchelli, Yin Kia Chiam, and 1 others. 2024a. Investigating efficacy of perplexity in detecting llm-generated code. *arXiv preprint arXiv:2412.16525*.

Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. 2024b. Licoeval: Evaluating llms on license compliance in code generation. *arXiv preprint arXiv:2408.02487*.

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2024c. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology*.

Zhenyu Xu and Victor S Sheng. 2024. Detecting ai-generated code assignments using perplexity of large language models. In *Proceedings of the aaai conference on artificial intelligence*, volume 38, pages 23155–23162.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025b. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Tong Ye, Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. 2024. Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. *arXiv preprint arXiv:2405.16133*.

A Other Metrics

A.1 Comments Ratio

$$CR = \frac{\text{Number of comment lines}}{\text{Total number of code lines}} \times 100\%.$$

A.2 Halstead Complexity Metrics

- Program Vocabulary: $n = n_1 + n_2$
- Program Length: $N = N_1 + N_2$
- Calculated Program Length: $\hat{N} = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2$
- Volume: $V = N \cdot \log_2 n$
- Difficulty: $D = \left(\frac{n_1}{2}\right) \cdot \left(\frac{N_2}{n_2}\right)$
- Effort: $E = D \cdot V$
- Time to Implement: $T = \frac{E}{18}$ (in seconds)
- Estimated Bugs: $B = \frac{E^{2/3}}{3000}$

where n_1 is the number of distinct operators, n_2 is the number of distinct operands, N_1 is the total number of operator occurrences, and N_2 is the total number of operand occurrences.

A.3 The maintainability index

- **Standard maintainability index**:

$$\text{MI}_{\text{std}} = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot \text{CC} - 16.2 \cdot \ln(\text{SLOC})$$

- **Custom maintainability index**:

$$\text{MI}_{\text{custom}} = \text{MI}_{\text{std}} + 50 \cdot \sin\left(\sqrt{2.4 \cdot \text{CR}}\right)$$

where V denotes the Halstead volume, CC is the cyclomatic complexity, SLOC is the number of logical source lines of code (i.e., executable statements excluding blank and comment lines, with multi-line statements counted as one), and CR is the comment ratio.

B Dataset

Table 3 shows the number of repositories in our dataset. Table 4 shows the mumber of problems per main algorithm across difficulty buckets.

B.1 Prompt

Figure 7, Figure 8 and Figure 9 show our prompt for generating and revising.

C Github Result

Figure 10- 17 show the comparison of variable, function, file, naming length and comment ratio (Python and C/C++) in LLM-generated vs. human-written code.

D Tags Frequencies

Table 6 shows the tag frequency of the collected questions. Table 7 shows the tag frequency results of the model output reasoning. Table 8 shows specific results at different levels of difficulty.

E Metrics Result

Tables 9 and 10 show the results containing reference-guided generation . Tables 11 and 12 show the results of large-scale model evaluation.

Quarter	Python-cs (#Repo / #Files)	Python-non-cs (#Repo / #Files)	C/C++-cs (#Repo / #Files)	C/C++-non-cs (#Repo / #Files)
2020 Q1	462 / 21643	139 / 3998	81 / 1532	29 / 385
2020 Q2	488 / 27190	111 / 2755	85 / 1002	18 / 241
2020 Q3	468 / 23008	152 / 5188	78 / 1077	18 / 206
2020 Q4	499 / 35083	131 / 3857	91 / 1367	19 / 490
2021 Q1	480 / 27155	139 / 3025	93 / 962	16 / 324
2021 Q2	523 / 28696	135 / 3471	117 / 1332	21 / 295
2021 Q3	520 / 27802	127 / 2448	121 / 1544	20 / 139
2021 Q4	508 / 29693	160 / 3649	105 / 1247	26 / 357
2022 Q1	486 / 24890	185 / 5684	160 / 1612	29 / 456
2022 Q2	498 / 31272	192 / 5495	170 / 2450	35 / 599
2022 Q3	495 / 34820	202 / 6358	208 / 3233	29 / 308
2022 Q4	515 / 37587	193 / 4873	218 / 2354	18 / 341
2023 Q1	521 / 45587	191 / 5862	225 / 2150	24 / 185
2023 Q2	508 / 38882	240 / 6345	226 / 2416	35 / 872
2023 Q3	506 / 36300	255 / 6110	267 / 2429	25 / 186
2023 Q4	525 / 44676	232 / 7428	277 / 3177	31 / 212
2024 Q1	524 / 34320	250 / 6581	266 / 3058	24 / 301
2024 Q2	533 / 44075	325 / 9628	320 / 4070	41 / 616
2024 Q3	530 / 48355	359 / 9620	366 / 5541	49 / 542
2024 Q4	529 / 40706	431 / 13031	405 / 4363	50 / 957
2025 Q1	520 / 59170	323 / 10541	296 / 4363	56 / 787
Total	10638 / 740910	4472 / 125947	4175 / 51279	613 / 8799

Table 3: Number of repositories and Python/C++ files per quarter and category

Main Algorithm	800–1199	1200–1599	1600–1999	2000+
implementation	25	13	6	2
brute force	10	9	7	12
constructive algorithms	5	6	9	7
greedy	5	8	4	1
math	3	2	3	4
binary search	1	4	8	9
dp	1	2	4	7
data structures	0	2	4	3
combinatorics	0	2	2	4
dfs and similar	0	2	3	1

Table 4: Number of problems per main algorithm across difficulty buckets.

Prompt

Your task is to carefully read the following problem description and implement a solution in {language}. Please first provide your reasoning in plain text, and then provide the corresponding code. Format your response as follows using Markdown:
Reasoning
<Please provide only your step-by-step reasoning in plain text here.>
Code
<Please provide only your code in {language} here, with no extra explanation or text.>
Here is the problem description: {context}

Figure 7: Prompt instructing LLMs to provide reasoning and solution code based only on problem descriptions.

Prompt

Your task is to carefully read the following problem description and revise the given code. The code given is AC code (correct and has passed the test.) Please first provide your reasoning in plain text, and then provide the corresponding code. Format your response as follows using Markdown:
Reasoning
<Please provide only your step-by-step reasoning in plain text here.>
Code
<Please provide only your code in {language} here, with no extra explanation or text.>
Here is the problem description: {context}
Here is the user's AC Code: {code}

Figure 8: Prompt instructing LLMs to provide reasoning process and solution code based on both problem descriptions and correct human-written code.

Prompt

Your task is to carefully read the following problem description and implement a solution in {language}. Return only the code without any explanations. Here is the problem description:\n\n{context}

Figure 9: Prompt instructing LLMs to provide only solution code based on problem descriptions.

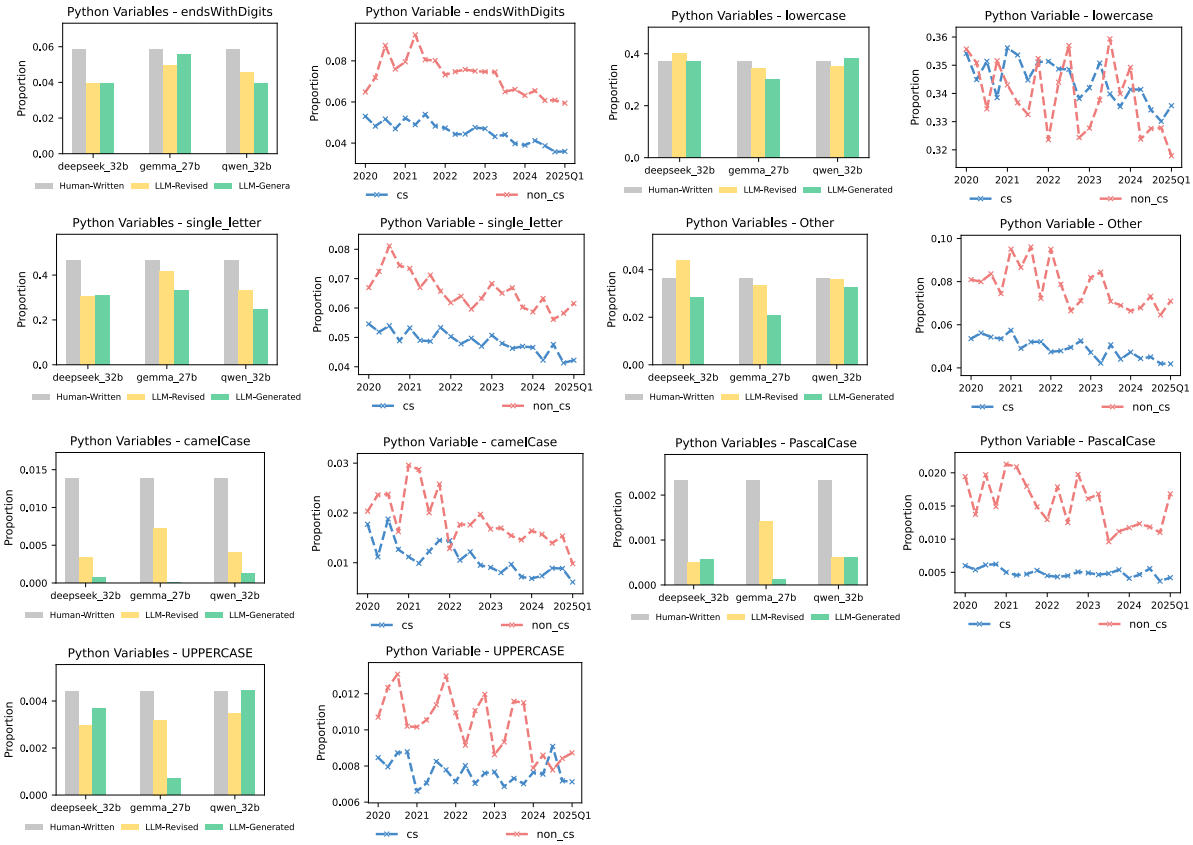


Figure 10: Comparison of **Python variable** naming styles in LLM-generated vs. human-written code and their temporal trends on GitHub.

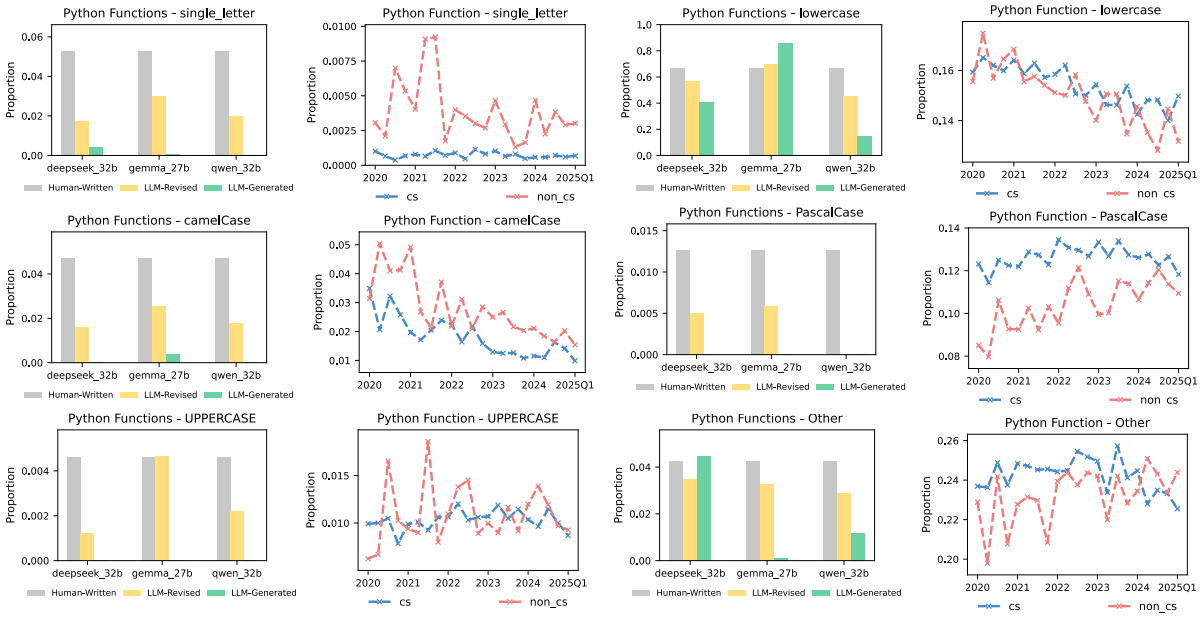


Figure 11: Comparison of **Python function** naming styles in LLM-generated vs. human-written code and their temporal trends on GitHub.

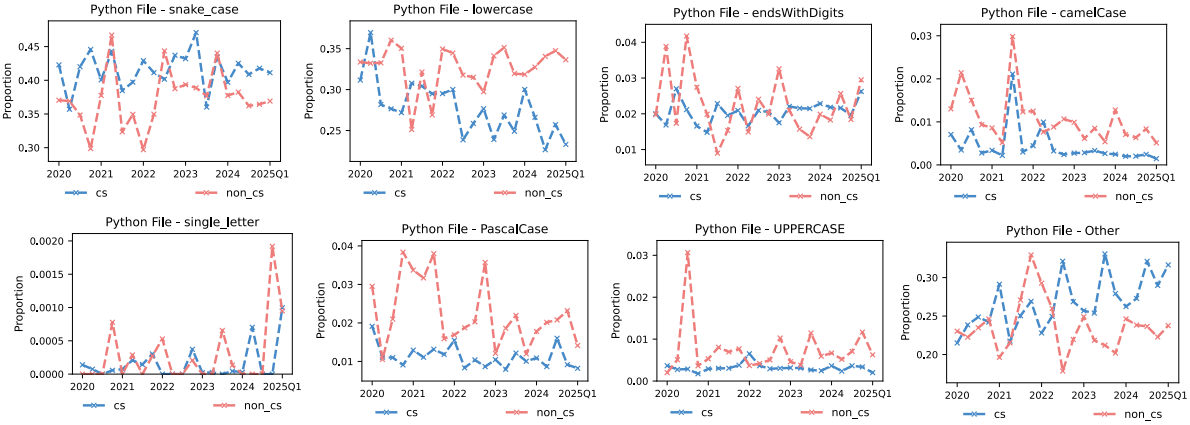


Figure 12: Comparison of **Python file** naming styles in LLM-generated vs. human-written code and their temporal trends on GitHub.

Model	**C/C++**			**Python**		
	AC vs ANS	AC vs REF	ANS vs REF	AC vs ANS	AC vs REF	ANS vs REF
Qwen-32B	0.2871	0.5884	0.3782	0.2761	0.5910	0.3340
Gemma-27B	**0.3012**	**0.7397**	0.3961	0.3703	0.7553	0.4198
DeepSeek-32B	0.1754	0.5204	0.2493	0.2474	0.4470	0.3323

Table 5: Jaccard similarity between the human-written code (AC), the initial LLM-generated code (ANS), and the LLM-rewritten code (REF) based on human-written code.

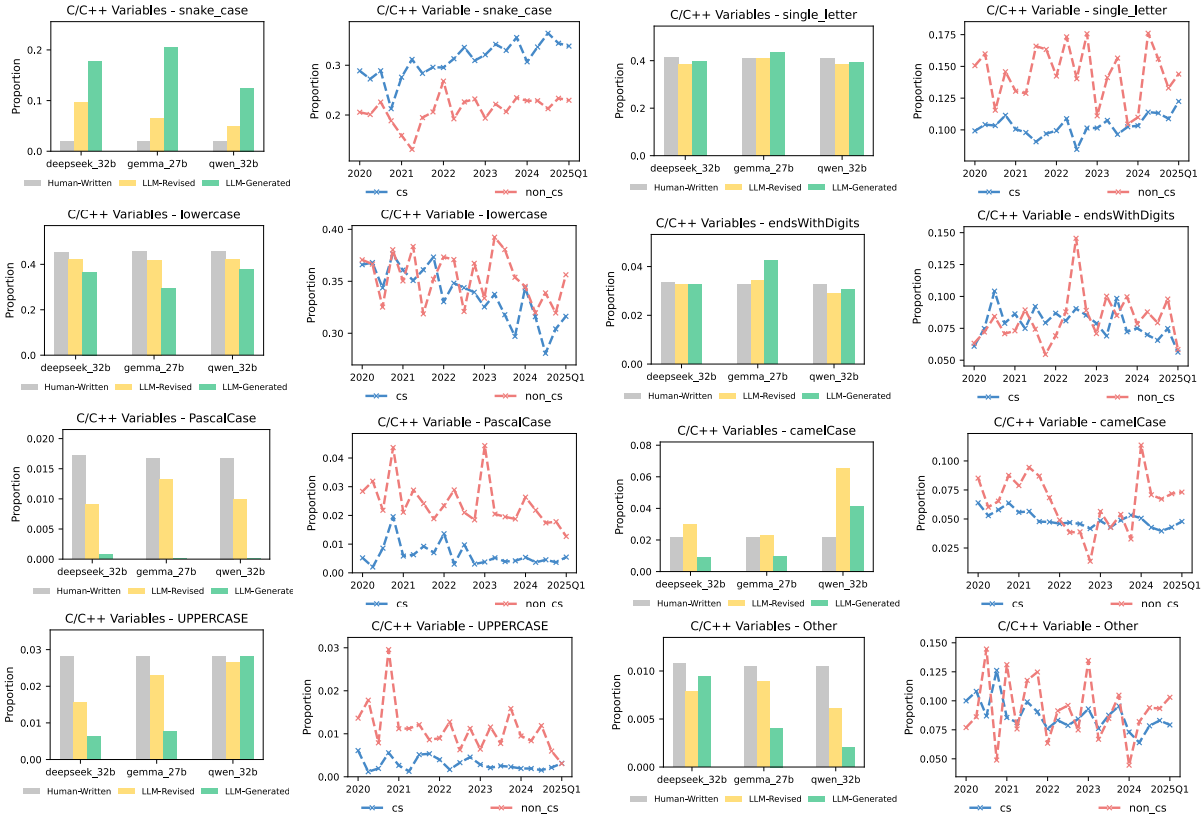


Figure 13: Comparison of **C/C++ variable** naming styles in LLM-generated vs. human-written code and their temporal trends on GitHub.

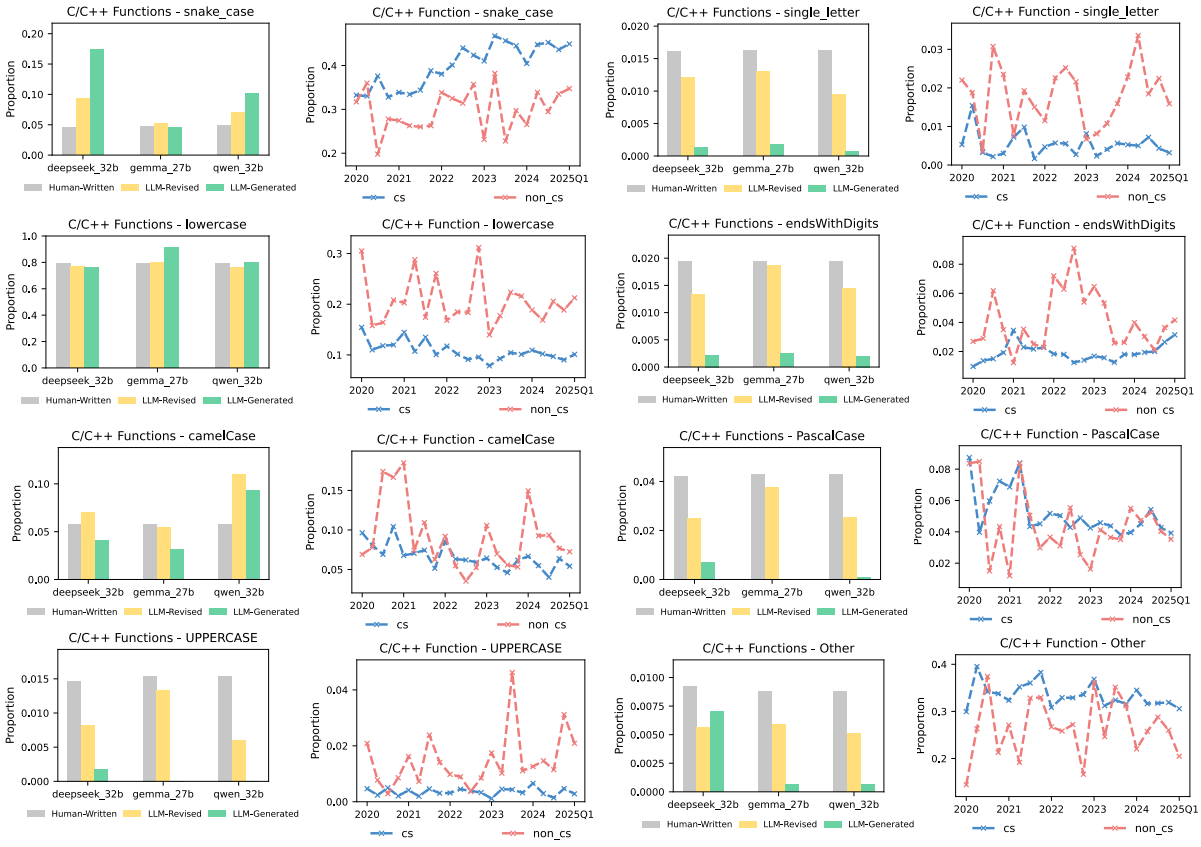


Figure 14: Comparison of **C/C++ function** naming styles in LLM-generated vs. human-written code and their temporal trends on GitHub.

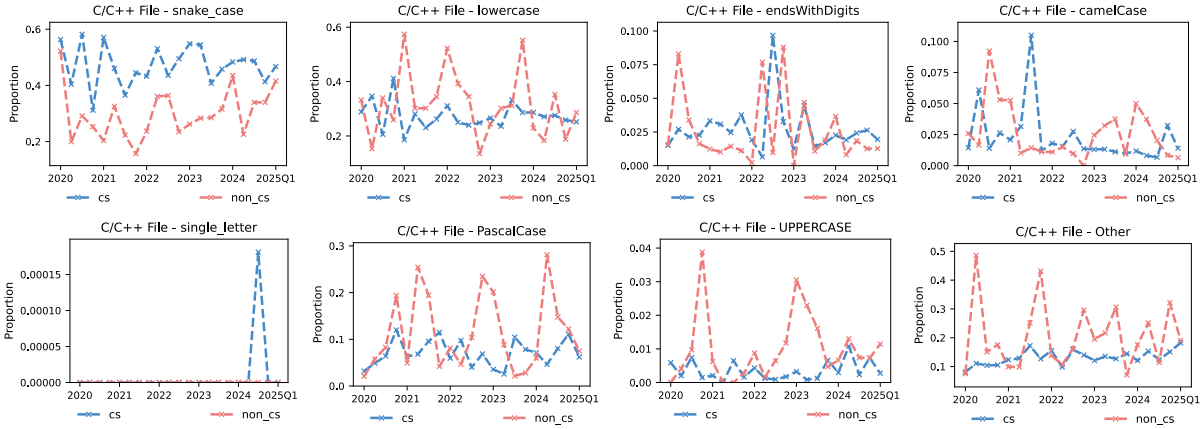


Figure 15: Comparison of **C/C++ file** naming styles in LLM-generated vs. human-written code and their temporal trends on GitHub.

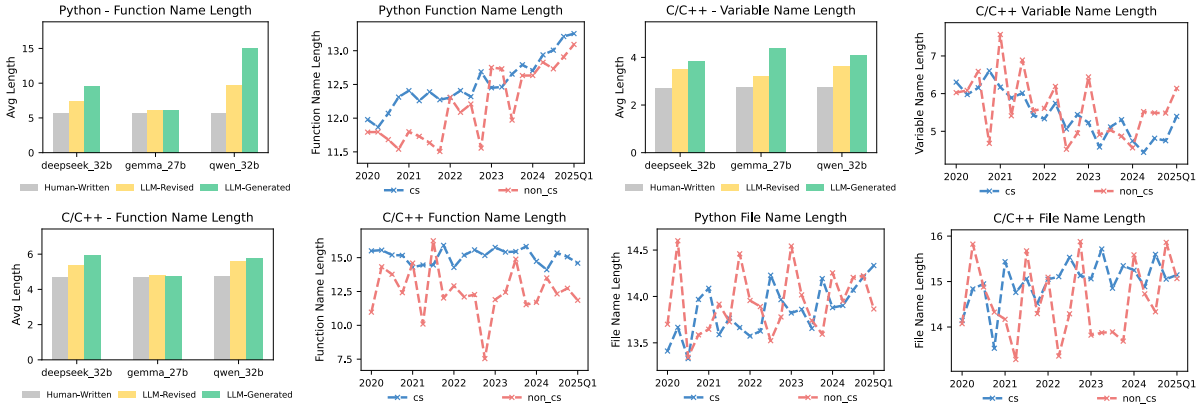


Figure 16: Comparison of naming length in LLM-generated vs. human-written code and their temporal trends on GitHub. We do not prompt the models to rewrite file names, so no comparison is made in this regard.

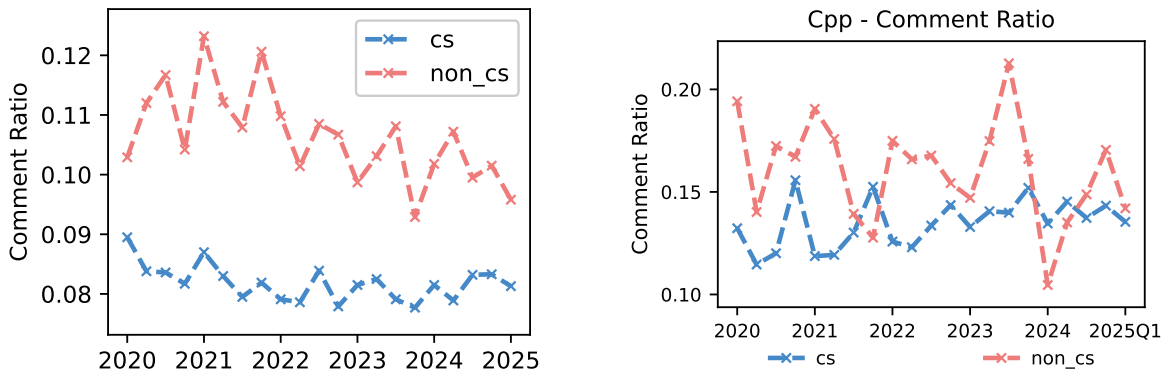


Figure 17: No consistent trend is observed in the comment ratio over time for either language.

Tag	Freq	Tag	Freq	Tag	Freq	Tag	Freq
2-sat	7	binary search	260	bitmasks	81	brute force	404
chinese remainder theorem	3	combinatorics	154	constructive algorithms	350	data structures	395
dfs and similar	260	divide and conquer	57	dp	565	dsu	91
expression parsing	26	fft	9	flows	34	games	62
geometry	157	graph matchings	18	graphs	312	greedy	513
hashing	55	implementation	849	interactive	20	math	631
matrices	39	meet-in-the-middle	10	number theory	184	probabilities	78
schedules	2	shortest paths	80	sortings	248	string suffix structures	25
strings	221	ternary search	15	trees	189	two pointers	113

Table 6: Frequencies of all algorithmic tags

Tag	A_q_p	R_q_p	A_q_c	R_q_c	A_g_p	R_g_p	A_g_c	R_g_c	A_d_p	R_d_p	A_d_c	R_d_c
2-sat	0	0	4	3	0	0	3	7	0	0	2	5
bfs	39	60	294	186	61	55	373	189	246	147	1781	384
binary search	138	114	286	451	34	125	89	352	274	165	1312	506
bitmasks	0	1	1	8	0	3	6	6	3	2	17	6
brute force	3	0	29	4	5	0	26	2	8	4	12	4
chinese remainder theo	0	0	1	0	0	0	0	0	0	0	6	0
combinatorics	2	2	8	1	0	0	1	5	4	4	25	8
constructive algorithms	0	0	0	0	0	0	0	0	0	0	0	0
data structures	36	15	134	202	8	8	54	158	27	31	162	243
dfs	47	58	441	579	74	72	490	719	32	42	374	489
divide and conquer	0	0	5	0	0	0	2	2	0	0	4	10
dp	346	26	1682	792	414	97	1801	777	1325	320	6388	993
dsu	1	6	24	12	11	9	5	52	40	7	135	38
expression parsing	0	0	0	0	0	1	0	0	0	0	0	0
fft	0	0	0	17	0	0	0	39	0	0	43	20
flows	0	0	3	1	1	0	17	1	3	1	64	8
games	29	5	24	18	37	23	53	22	16	8	54	25
geometry	7	3	21	7	0	0	1	3	1	6	30	9
graph matchings	0	0	0	0	0	0	0	0	0	0	0	0
graphs	7	2	35	13	1	4	10	16	21	4	77	42
greedy	72	18	195	37	81	9	270	48	61	23	221	77
hashing	0	2	7	31	0	0	0	40	2	4	47	41
implementation	205	83	610	276	14	76	68	290	49	61	244	294
interactive	1	2	4	3	3	2	7	10	3	1	0	2
math	8	27	0	7	46	18	0	3	117	10	25	2
matrices	2	5	11	10	1	0	10	7	6	7	85	19
meet-in-the-middle	0	0	2	2	0	0	0	0	0	0	1	3
number theory	0	0	4	2	0	0	1	0	1	0	3	1
probabilities	16	23	88	98	9	10	88	85	9	9	267	81
schedules	6	2	3	6	0	2	3	2	2	2	6	2
shortest paths	8	5	72	48	8	11	85	49	5	5	195	50
similar	17	12	61	45	18	14	53	49	87	27	922	43
sort	241	60	548	147	301	75	569	153	510	173	1665	240
string	807	509	1417	894	1094	751	1874	1184	2369	999	5863	1203
strings	162	153	375	245	230	158	507	288	248	153	800	241
suffix	46	36	147	146	82	39	93	122	267	58	955	206
ternary search	0	0	0	19	0	0	3	13	1	0	36	7
trees	22	18	127	111	22	15	133	158	36	47	699	194
two pointers	26	13	54	15	8	4	10	7	14	6	33	20

Table 7: Frequencies of each output tag across different generation types. The format `T_M_L` denotes **Type** (A: ANS, R: REF), **Model** (q: Qwen, g: Gemma, d: Deepseek), and **Language** (p: Python, c: C/C++).

Model-Language-Type	800–1199		1200–1599		1600–1999		2000+	
	Match Rate	Error Rate	Match Rate	Error Rate	Match Rate	Error Rate	Match Rate	Error Rate
gemma-py-ans	5.61%	10.20%	10.26%	16.78%	16.15%	20.77%	13.51%	22.97%
gemma-py-ref	5.61%	8.16%	9.32%	11.19%	17.44%	16.92%	16.89%	26.35%
gemma-cpp-ans	5.69%	13.37%	11.68%	16.60%	18.45%	18.74%	22.40%	26.14%
gemma-cpp-ref	7.67%	9.16%	14.34%	16.60%	28.26%	19.77%	33.72%	32.89%
Qwen-py-ans	9.44%	18.11%	16.08%	22.38%	22.56%	32.31%	27.70%	32.43%
Qwen-py-ref	9.44%	11.22%	9.79%	16.08%	15.38%	20.51%	15.54%	23.65%
Qwen-cpp-ans	13.37%	20.54%	20.08%	24.18%	31.48%	36.75%	32.97%	45.21%
Qwen-cpp-ref	8.91%	11.88%	13.93%	19.47%	27.23%	22.11%	36.30%	32.64%
DeepSeek-py-ans	5.87%	13.01%	11.19%	17.02%	13.85%	15.38%	14.19%	18.24%
DeepSeek-py-ref	6.12%	10.97%	11.89%	14.22%	17.69%	17.44%	19.59%	24.32%
DeepSeek-cpp-ans	9.70%	18.33%	15.57%	25.44%	23.85%	28.44%	22.22%	32.04%
DeepSeek-cpp-ref	6.74%	12.67%	15.57%	17.32%	31.12%	24.80%	38.15%	33.61%

Table 8: Match rate and error rate for questions across different difficulty levels.

Model	h1	h2	N1	N2	Vocab	Length	Cal_Len	Volume	Difficul	Effort	Time_Sec	Bugs
AC-Python	5.39	17.53	14.80	28.76	22.92	43.56	93.63	216.66	4.41	1466.48	81.47	0.07
q-ANS-Python	4.91	14.66	11.93	23.15	19.57	35.08	74.64	164.70	3.87	939.75	52.21	0.05
q-REF-Python	4.97	14.45	11.61	22.52	19.42	34.14	73.82	160.69	3.88	946.77	52.60	0.05
g-ANS-Python	4.94	15.82	12.93	25.62	20.76	38.55	81.57	185.91	3.94	1083.57	60.20	0.06
g-REF-Python	4.79	14.00	11.38	22.13	18.79	33.51	71.37	157.69	3.81	947.36	52.63	0.05
d-ANS-Python	3.33	8.90	7.47	14.40	12.23	21.87	45.19	101.22	2.68	631.26	35.07	0.03
d-REF-Python	5.28	15.60	13.36	25.72	20.88	39.08	84.08	192.76	4.35	1367.70	75.98	0.06
AC-C/C++	10.33	79.36	140.95	534.78	89.70	675.73	555.79	4597.99	33.53	227661.62	12647.87	1.53
q-ANS-C/C++	9.15	55.84	82.68	276.47	64.98	359.15	358.02	2216.74	22.16	66253.53	3680.75	0.74
q-REF-C/C++	9.66	58.73	95.64	367.51	68.38	463.15	383.55	2930.18	28.79	126870.70	7048.37	0.98
g-ANS-C/C++	8.02	47.59	98.85	272.75	55.61	371.60	292.60	2205.17	22.62	67722.90	3762.38	0.74
g-REF-C/C++	9.50	68.39	112.94	428.06	77.89	541.00	466.15	3589.22	28.44	153739.81	8541.10	1.20
d-ANS-C/C++	4.78	29.71	50.56	170.63	34.49	221.19	188.31	1368.60	13.08	49152.88	2730.72	0.46
d-REF-C/C++	9.44	64.40	108.91	417.83	73.84	526.74	431.17	3439.53	29.10	158810.53	8822.81	1.15

Table 9: Halstead results. Each label follows the format *model_type_language*, where *type* refers to the experimental setting and *language* indicates the programming language. In the model abbreviation, `q`, `g`, and `d` refer to Qwen, Gemma, and DeepSeek, respectively. Metric abbreviations are as follows: `cal_len` = calculated program length, `difficul` = difficulty, `time_sec` = time to implement, `bugs` = estimated bugs.

Model	Volume	Cyclomatic	SLOC	LLOC	Comment_Rate	mi_std	mi_custom
AC-Python	216.66	2.60	20.44	21.23	3.96	62.41	62.87
Qwen-ANS-Python	164.70	6.25	20.98	21.43	22.78	78.56	78.58
Qwen-REF-Python	160.69	2.68	17.38	17.66	6.03	66.56	67.07
gemma-ANS-Python	185.91	8.37	23.37	23.43	1.18	58.43	58.41
gemma-REF-Python	157.69	2.66	17.81	18.23	2.19	64.18	64.63
DeepSeek-ANS-Python	101.22	1.05	14.07	14.16	2.01	76.16	76.66
DeepSeek-REF-Python	192.76	3.24	21.96	22.15	7.88	67.82	68.40
AC-C/C++	4597.99	15.94	69.39	49.72	0.05	41.96	35.51
Qwen-ANS-C/C++	2216.74	10.67	42.34	37.30	0.04	48.51	41.59
Qwen-REF-C/C++	2930.18	13.19	50.31	42.20	0.01	41.80	39.69
gemma-ANS-C/C++	2205.17	14.47	46.66	42.20	0.01	41.74	40.27
gemma-REF-C/C++	3589.22	14.16	58.18	44.73	0.02	42.37	38.26
DeepSeek-ANS-C/C++	1368.60	6.37	29.11	24.45	0.01	68.15	66.52
DeepSeek-REF-C/C++	3439.53	14.61	59.73	49.05	0.03	43.85	39.44

Table 10: Maintainability results. Each label follows the format *model_type_language*, where *type* refers to the experimental setting and *language* indicates the programming language. Metric abbreviations are as follows: `mi_std` = standard maintainability index, `mi_custom` = custom maintainability index.

Model	Language	Stat.	h1	h2	N_1	N_2	Vocab	Length	Cal_Len	Volume	Difficul	Effort	Time_Sec	Bugs
Claude	Python	mean	6.22	20.36	18.01	35.13	26.58	53.14	114.86	277.88	5.30	2303.85	127.99	0.09
		std	1.31	8.14	8.26	16.27	9.07	24.51	58.33	164.09	1.49	2061.31	114.52	0.05
	C/C++	mean	8.54	65.90	96.68	312.96	74.44	409.64	430.84	2611.66	19.76	67162.33	3731.24	0.87
		std	0.91	11.60	23.72	75.44	11.94	96.36	89.76	698.95	4.31	34582.01	1921.22	0.23
DSV3	Python	mean	6.10	18.37	17.23	33.28	24.46	50.52	100.73	254.75	5.49	2102.24	116.79	0.08
		std	0.77	4.14	4.58	8.93	4.65	13.50	28.68	85.88	0.99	1111.11	61.73	0.03
	C/C++	mean	7.65	48.08	71.45	249.61	55.73	321.06	294.59	1908.68	19.52	51055.60	2836.42	0.64
		std	0.62	3.52	10.80	36.49	3.81	46.29	26.88	305.03	2.99	18789.07	1043.84	0.10
DSR1	Python	mean	6.84	22.29	20.54	39.59	29.12	60.13	128.84	319.45	6.05	2902.90	161.27	0.11
		std	1.23	7.26	7.64	14.87	8.13	22.49	51.49	147.19	1.47	1927.68	107.09	0.05
	C/C++	mean	8.42	57.94	84.00	337.67	66.36	421.67	377.04	2720.12	22.66	98184.66	5454.70	0.91
		std	0.88	18.49	31.58	184.28	18.88	213.40	158.33	1753.39	5.24	102703.88	5705.77	0.58
Gemma	Python	mean	5.52	21.20	17.90	36.14	26.72	54.04	115.75	278.83	4.60	1792.06	99.56	0.09
		std	0.92	6.32	5.92	12.28	6.86	18.17	42.99	117.70	0.98	1076.68	59.82	0.04
	C/C++	mean	7.11	47.51	112.19	293.85	54.62	406.04	287.21	2379.33	21.96	67879.88	3771.10	0.79
		std	0.65	3.51	24.80	53.41	3.81	76.19	26.26	474.05	4.14	26880.38	1493.35	0.16
GPT	Python	mean	5.01	13.60	11.32	22.04	18.62	33.36	67.50	151.65	4.07	912.18	50.68	0.05
		std	1.02	3.80	3.58	7.07	4.53	10.64	23.91	59.85	1.07	528.82	29.38	0.02
	C/C++	mean	7.59	48.34	63.12	203.35	55.93	266.48	295.94	1571.22	15.88	30078.52	1671.03	0.52
		std	0.93	7.75	11.30	32.64	8.08	41.75	57.16	283.63	2.99	9882.98	549.05	0.09
Llama	Python	mean	5.80	17.89	15.24	29.68	23.69	44.93	95.89	222.09	4.78	1582.12	87.90	0.07
		std	1.27	6.22	6.22	12.21	7.12	18.42	41.73	112.55	1.44	1201.02	66.72	0.04
	C/C++	mean	7.76	47.56	71.16	244.13	55.31	315.29	291.68	1869.23	19.49	52663.32	2925.74	0.62
		std	1.09	7.90	21.61	70.48	8.37	88.71	60.37	579.59	5.68	49067.76	2725.99	0.19
Qw4B	Python	mean	5.20	15.25	12.87	24.97	20.45	37.84	78.39	178.54	4.27	1105.91	61.44	0.06
		std	0.85	4.14	3.87	7.66	4.77	11.52	28.29	70.75	0.91	651.17	36.18	0.02
	C/C++	mean	7.47	50.81	66.06	235.00	58.28	301.07	315.46	1822.14	16.76	41082.45	2282.36	0.61
		std	0.64	7.44	12.31	48.69	7.68	59.78	58.12	424.64	2.86	18516.05	1028.67	0.14
Qw8B	Python	mean	5.11	15.05	12.86	24.96	20.16	37.82	77.21	180.06	4.19	1174.21	65.23	0.06
		std	1.09	4.86	4.92	9.69	5.65	14.60	32.15	86.64	1.21	852.52	47.36	0.03
	C/C++	mean	7.29	47.40	64.41	223.15	54.69	287.56	289.40	1719.90	16.42	42327.90	2351.55	0.57
		std	0.91	7.83	18.62	71.91	8.26	89.09	60.75	643.36	3.90	43502.31	2416.80	0.21
Qw14B	Python	mean	5.64	16.86	14.27	27.67	22.50	41.93	88.48	201.97	4.62	1330.55	73.92	0.07
		std	0.90	4.24	4.16	8.17	4.90	12.32	28.49	74.61	0.99	763.02	42.39	0.02
	C/C++	mean	7.78	55.85	72.02	255.75	63.62	327.77	354.15	2021.52	17.46	46444.34	2580.24	0.67
		std	0.73	8.96	14.89	53.41	9.29	67.10	70.76	482.39	3.33	23781.57	1321.20	0.16
Qw32B	Python	mean	5.90	17.56	14.72	28.50	23.46	43.22	93.84	211.90	4.76	1458.62	81.03	0.07
		std	1.14	5.01	4.87	9.50	5.83	14.35	33.83	87.12	1.22	921.81	51.21	0.03
	C/C++	mean	8.04	54.25	72.48	260.03	62.29	332.51	342.14	2038.57	18.82	51082.68	2837.93	0.68
		std	0.89	8.35	15.50	56.92	8.73	70.74	64.18	488.85	4.02	23772.82	1320.71	0.16
QwCo	Python	mean	5.51	17.09	14.74	28.57	22.60	43.31	90.83	212.60	4.58	1457.38	80.97	0.07
		std	1.27	6.30	6.22	12.17	7.22	18.37	42.43	112.54	1.45	1142.83	63.49	0.04
	C/C++	mean	7.70	43.79	64.09	216.48	51.49	280.57	264.30	1633.43	18.53	41956.16	2330.90	0.54
		std	1.04	4.85	16.98	57.39	5.31	72.14	36.29	455.12	5.03	27470.03	1526.11	0.15
Human	Python	mean	5.89	20.20	17.30	33.84	26.09	51.14	111.90	263.59	4.96	1973.81	109.66	0.09
		std	0.83	4.00	3.72	7.41	4.58	11.13	25.66	66.45	0.85	624.38	34.69	0.02
	C/C++	mean	9.66	67.54	116.54	387.92	77.20	504.47	454.55	3303.25	26.78	145862.09	8103.45	1.10
		std	1.69	31.47	98.16	254.47	32.55	340.85	259.09	2649.44	10.24	235927.01	13107.06	0.88

Table 11: Halstead results on the evaluation subset. Model names use the following abbreviations: `Claude` = Claude-3.5-Sonnet, `DSV3` = DeepSeek-V3, `DSR1` = DeepSeek-R1, `Gemma` = Gemma-3-27B, `GPT` = GPT-4o-mini, `Llama` = Llama 3.3 Nemotron Super 49B v1, `Qw14B` = Qwen3-14B, `Qw32B` = Qwen3-32B, `Qw4B` = Qwen3-4B, `Qw8B` = Qwen3-8B, and `QwCo` = Qwen2.5-Coder-32B-Instruct. The label `Human` refers to human-written code. Metric abbreviations are: `Cal_Len` = calculated program length, `Difficul` = difficulty, `Time_Sec` = time to implement, `Bugs` = estimated bugs.

Model	Language	Stat.	Volume	Cyclomatic	SLOC	LLOC	Comment_Rate	MI_Std	MI_Custom
Claude	Python	mean	277.88	9.37	29.06	29.22	20.47	72.62	72.72
		std	164.09	4.71	9.71	9.70	11.13	7.45	7.45
	C/C++	mean	2611.66	11.83	49.28	40.61	0.09	53.70	39.47
		std	698.95	3.55	10.44	9.13	0.05	5.63	2.71
DSV3	Python	mean	254.75	4.16	27.69	27.88	8.96	63.28	63.95
		std	85.88	1.91	5.54	5.59	8.35	4.72	4.68
	C/C++	mean	1908.68	10.37	42.54	37.72	0.00	42.87	42.02
		std	305.03	2.10	5.73	5.55	0.01	2.06	1.59
DSR1	Python	mean	319.45	5.63	33.16	33.62	18.92	56.58	57.29
		std	147.19	3.53	9.56	9.67	47.74	7.55	7.49
	C/C++	mean	2720.12	12.13	48.28	41.80	0.02	42.08	40.41
		std	1753.39	3.52	12.44	9.98	0.05	3.89	2.91
Gemma	Python	mean	278.83	11.64	31.14	31.49	0.54	52.74	52.74
		std	117.70	3.76	6.82	6.89	0.63	3.80	3.79
	C/C++	mean	2379.33	18.64	51.23	47.04	0.00	38.88	38.23
		std	474.05	5.14	8.57	8.60	0.00	2.89	2.51
GPT	Python	mean	151.65	4.34	19.77	20.18	7.35	66.81	67.11
		std	59.85	2.38	3.89	3.99	8.25	8.40	8.40
	C/C++	mean	1571.22	8.01	33.61	28.85	0.02	48.89	44.51
		std	283.63	1.75	4.63	4.22	0.03	5.41	1.87
Llama	Python	mean	222.09	6.67	24.25	24.67	3.60	59.23	59.45
		std	112.55	3.85	7.10	7.27	7.10	7.05	7.04
	C/C++	mean	1869.23	10.02	39.97	34.97	0.00	43.38	42.66
		std	579.59	3.80	9.92	9.21	0.01	3.50	3.05
Qw4B	Python	mean	178.54	5.43	25.25	25.61	14.78	66.60	66.90
		std	70.75	1.80	4.61	4.63	11.42	5.08	5.07
	C/C++	mean	1822.14	9.85	38.15	33.42	0.02	46.83	43.38
		std	424.64	2.20	6.74	5.96	0.02	3.38	2.02
Qw8B	Python	mean	180.06	4.63	23.17	23.57	6.59	62.92	63.26
		std	86.64	2.23	5.84	5.91	8.36	6.05	6.03
	C/C++	mean	1719.90	8.72	36.59	31.88	0.02	46.74	44.08
		std	643.36	3.47	9.41	8.86	0.03	3.89	2.63
Qw14B	Python	mean	201.97	5.68	27.68	28.20	9.60	64.48	64.77
		std	74.61	1.97	5.15	5.20	7.13	4.72	4.72
	C/C++	mean	2021.52	9.74	41.83	36.14	0.03	47.95	42.10
		std	482.39	2.37	7.65	6.87	0.03	3.94	2.15
Qw32B	Python	mean	211.90	4.71	26.86	27.27	4.78	61.56	62.00
		std	87.12	2.45	6.03	6.09	4.69	5.79	5.76
	C/C++	mean	2038.57	10.26	41.78	37.35	0.02	45.98	42.01
		std	488.85	2.73	8.41	8.02	0.02	4.43	2.52
QwCo	Python	mean	212.60	6.19	23.45	23.93	1.26	57.81	57.94
		std	112.54	3.00	7.00	7.13	3.01	5.88	5.90
	C/C++	mean	1633.43	9.22	34.53	30.81	0.00	44.76	44.44
		std	455.12	2.97	8.02	7.71	0.00	3.13	2.84
Human	Python	mean	263.59	2.64	36.01	23.59	6.51	60.82	60.82
		std	66.45	1.45	9.26	4.76	11.64	5.64	5.64
	C/C++	mean	3303.25	12.62	55.01	39.09	0.05	45.10	38.82
		std	2649.44	7.00	29.40	18.13	0.09	7.81	6.46

Table 12: Maintainability results on the evaluation subset, broken down by model, language (Python vs. C/C++), and statistic (mean vs. std). Model abbreviations follow those in Table 11