Revisiting the Impact of Pursuing Modularity for Code Generation

Anonymous ACL submission

Abstract

Modular programming, which aims to construct the final program by integrating smaller, independent building blocks, has been regarded as a desirable practice in software development. However, with the rise of recent code generation agents built upon large language models (LLMs), a question emerges: is this traditional practice equally effective for these new tools? In this work, we assess the impact of modularity in code generation by introducing a novel metric for its quantitative measurement. Surprisingly, unlike conventional wisdom on the topic, we find that modularity is not a core factor for improving the performance of code generation models. We also explore potential explanations for why LLMs do not exhibit a preference for modular code compared to non-modular code.

1 Introduction

With recent advances in the capabilities of large language models (LLMs; OpenAI, 2024; Gemini Team, 2024; *inter alia*), their application areas have expanded beyond simple text-based tasks. Among these, coding assistants are becoming practically essential for programmers, enhancing their efficiency through tasks such as natural language to code (NL2Code) generation.

Similar to other use cases of LLMs, coding assistants are typically utilized in zero- or few-shot manners, without task-specific fine-tuning. The problem is that as the length of code is usually much longer than that of a sentence, the number of code examples available for each run is strictly limited. Furthermore, the same functionality can be represented with different forms of code, making it challenging for users to select a proper example for a target task. It is thus important to understand what characteristics of the code provided to the agents contribute to the final performance of such models. Among the many possible properties that influence the characteristics of code snippets, this work investigates the impact of **code modularity** on the performance of LLMs for NL2Code generation.

Modular programming, the practice of building software with independent components, has long been considered a cornerstone of good software development. While this paradigm facilitates desirable properties of code for *human programmers*, such as reusability, readability, and maintainability, it remains an open question whether it offers the same level of effectiveness for *LLMs*.

Notably, Jain et al. (2024) argued that leveraging a set of modular functions can improve code generation accuracy for both in-context learning (ICL) and fine-tuning. As it is not trivial to guarantee the modularity of each code snippet, the authors asked GPT-3.5-Turbo[1] to convert an existing code snippet into a more modular one, while ensuring its functional correctness.

However, we claim that their report warrants revisiting for two reasons. First, since LLMs are known for their verbosity, it is unclear whether the conversion process aimed solely for modularity or inadvertently introduced unexpected side effects. Second, the lack of a formally defined quantitative method for estimating modularity hinders more extensive analyses related to the problem.

In this paper, we (re-)investigate the effectiveness of pursuing modularity in NL2Code generation. We aim to push the boundaries of previous work by (1) introducing a novel metric that quantifies the modularity of a code snippet using numeric values. Based on the metric, we (2) classify code snippets as modular or non-modular without relying on LLMs, and evaluate how each category contributes to performance.[2] Moreover, beyond previous work, we (3) conduct experiments on models with parameters exceeding 7B (i.e., 33B and 34B)

[1] https://platform.openai.com/docs/models/gpt-3-5-turbo

[2] Note that this was infeasible in the previous study (Jain et al., 2024) as there was no clear standard for determining whether each code snippet is modular or not.

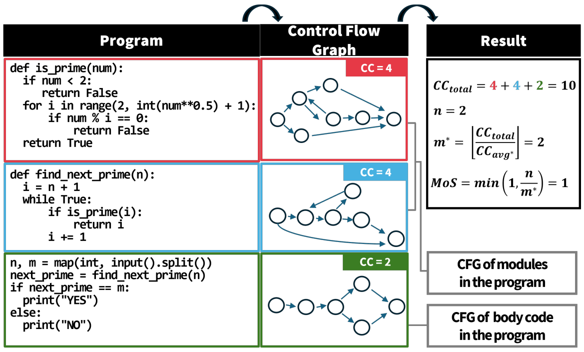


Figure 1: Illustration of Code Complexity (CC) and Modularity Score (MoS) computation. We first build control flow graphs from the given code to derive CC values. The CC values are then used to calculate intermediate values, e.g., m, which are ultimately used to derive MoS based on its definition involving n and m^*.

to investigate the impact of model scale.

In experiments, we discover that contrary to conventional wisdom in the literature, **the modularity of a code example may not be the crucial factor for performance**. We also explore potential explanations for why LLMs do not exhibit a preference for modular code compared to non-modular code.

2 Quantitative Definition of Modularity

To assess the impact of code modularity, the first essential step is to develop a method that provides a measurable score for code modularity. While the previous study (Jain et al., 2024) bypassed this vital step,[3] we present a reasonable metric for estimating code modularity, which is challenging due to the inherent subjectivity of the concept itself.

Inspired by the software engineering literature, we employ **Cyclomatic Complexity (CC)** (McCabe, 1976) to determine the ideal number of modules, m^*, for a given code snippet. CC is calculated as E - N + 2, where E and N correspond to the number of edges and nodes, in a control-flow graph representation of the target code. CC can be computed at either the whole code level (total CC; CC_{total}) or the function level (meaning the average CC across all functions in the code; CC_{avg}).

A high CC value generally indicates a complex code structure. It functions as a guideline for code decomposition, suggesting that a function whose CC is exceeding a certain threshold value τ, e.g., 5 (McCabe, 1976) or 10 (McConnell, 2004), might benefit from being broken down into smaller sub-functions. Based on the concept, we assume that the average CC of an ideal modular code example, denoted by CC_{avg^*}, should be equal to the threshold τ.[4] In other words, ideally, every function within a modular code snippet is expected to have a CC value of τ. Following the intuition, we define m^*, the number of ideal modules, as follows:

$$m^* = \lfloor \frac{\text{CC}_{\text{total}}}{\text{CC}_{\text{avg}^*}} \rfloor = \lfloor \frac{\text{CC}_{\text{total}}}{\tau} \rfloor,$$

Finally, we define the modularity score, dubbed MoS, as follows:

$$\text{MoS} = \begin{cases} \min\left(1, \frac{n}{m^*}\right) & \text{if } m^* > 0 \\ 0 & \text{if } m^* = n = 0 \\ 1 & \text{if } m^* = 0,\ n > 0 \end{cases},$$

where n is equal to the actual number of modules in the target code. That is, the closer n (actual number of modules) is to m^* (ideal number of modules), the higher the modularity is considered to be.[5]

3 Four Code Categories by Modularity

This allows us to create four distinct clusters of code separated by their modularity levels. Figure 2 in Appendix illustrates example code from each category for the same problem.

Modular Code (MC) is a collection of code snippets with the highest MoS among solutions for each problem in a dataset.

Singular Code (SC) represents another set of solution code examples for the same problems corresponding to **MC**, with MoS being 0.

Transformed Modular Code (TMC) can be obtained by utilizing GPT-3.5-Turbo (f) to transform **SC** into code with high MoS. The conversion process can be represented by the following:

$$\textbf{TMC} = f(I, Q, \textbf{SC}),$$

where I represents a transformation instruction and Q is the problem description of **SC**.[6]

[3]The authors instead utilized LLMs to transform all code snippets into supposedly modular ones.

[4]Given two choices for τ, i.e., 5 or 10, we set τ to 5 to encourage a sparser distribution of modularity scores (MoS).

[5]In extreme cases where $m = 0$ (no modularization required), the modularity score is set to 0 if no actual modules are used ($n = 0$) and 1 otherwise ($n > 0$).

[6]See Figure 3 for prompt details on the conversion process.

Model	Size	Code Type	Introductory		Interview		Competition		Average	
			pass@1	pass@5	pass@1	pass@5	pass@1	pass@5	pass@1	pass@5
Code Llama	7B	TMC	**14.67**	**19.63**	**2.28**	**3.98**	**0.21**	**0.59**	**4.45**	**6.66**
		TSC	13.84	17.15	2.16	3.61	0.07	0.24	4.20	6.07
DeepSeekCoder	6.7B	TMC	**34.26**	**40.74**	**9.60**	**13.41**	**0.76**	**1.93**	**13.49**	**17.63**
		TSC	33.24	39.73	8.55	12.40	0.55	1.21	12.55	16.64

Table 1: Results on APPS measured by pass@k. We use n=10 for pass@1 and pass@5. The best results are in **bold** for each section. Two-shot prompting is applied to generate code. **Code Type** refers to two distinct groups of code used for demonstrations. We find that TMC slightly outperforms TSC but the performance gaps are insignificant.

Model	Size	Code Type	CodeContests	
			pass@1	pass@10
Code Llama	7B	MC	1.98	8.02
		SC	2.58	8.81
		TMC	2.57	10.18
		TSC	**4.35**	**10.67**
	34B	MC	4.11	12.78
		SC	**5.83**	14.1
		TMC	3.39	13.55
		TSC	5.61	**15.32**
DeepSeekCoder	6.7B	MC	5.3	12.78
		SC	7.15	16.27
		TMC	8.02	**17.88**
		TSC	**8.19**	17.79
	33B	MC	6.79	16.14
		SC	8.87	20.5
		TMC	**9.38**	**22.74**
		TSC	8.78	22.09

Table 2: Results on CodeContests measured by pass@k. We use n=10 for pass@1 and n=50 for pass@5, respectively. The best results are in **bold** for each section. Two-shot prompting is applied for generating code given natural language queries. **Code Type** refers to four distinct groups of code used for demonstrations. We reveal no significant impact of code type on performance.

Transformed Singular Code (TSC) is a variation from **TMC**, whose modularity is manually removed by human programmers. By minimizing the influence of factors other than modularity through the comparison of **TSC** and **TMC**, it encourages a rigorous evaluation of the impact of modularity.

4 Experimental Setups

We explore the impact of modularity by comparing how the four code collections, categorized by their modularity levels, affect performance. To mimic real-world usage, we focus on the case of utilizing code LLMs with few-shot in-context learning. We leverage two-shot demonstrations (providing two code examples) unless otherwise specified.[7] We will open-source our code after the review process.

Models. We use two LLMs for code generation—Code Llama (7B, 34B; Rozière et al., 2024) and DeepSeekCoder (6.7B, 33B; Guo et al., 2024).

Datasets. We employ two NL2Code generation datasets—APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022).[8] They are based on competitive programming contests and provide a set of different solutions for each programming problem. For each dataset, the groups of **MC** and **SC** demonstrations are chosen from solutions for randomly selected problems. **SC** examples are then converted into **TMC**, and finally, **TSC** is manually obtained. In this study, we focus our evaluation on Python.

Evaluation Metrics. We apply an unbiased version of pass@k (Chen et al., 2021), which measures the functional correctness of generated programs by running them against test cases. For each problem, LLMs are prompted to generate n programs, and we determine c, the number of programs that pass the test cases. In addition, k ($k \leq n$) specifies the granularity of evaluation such that the metric indicates the probability of finding at least one correct solution when sampling k programs out of the n generated ones. The metric is then averaged over all problems. As a result, pass@k is computed as:

$$\text{pass@}k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right].$$

5 Main Results

Table 1 and Table 2 present results on APPS and CodeContests, categorized by the modularity of the

[7] Refer to Figure 4 and Figure 5 for prompt details.

[8] Note that representative code generation benchmarks, e.g., HumanEval (Chen et al., 2021), typically provide code snippets whose length restricts the possibility of modularization.

Model	Size	Pearson	Spearman
Code Llama	7B	-0.34 (0)	-0.31 (0)
DeepSeekCoder	6.7B	-0.21 (0.04)	-0.25 (0.01)

Table 3: Correlations between modularity (MoS) and performance (pass@1), evaluated on CodeContests. They consistently show weak negative relationships. Numbers in parentheses represent p-values.

Model	Size	$P(\mathcal{C}_{\text{SC}}\|\mathcal{D}) \Uparrow$	$P(\mathcal{C}_{\text{MC}}\|\mathcal{D}) \Uparrow$
Code Llama	7B	**59.4**	40.6
	34B	**52.5**	47.5
DeepSeekCoder	6.7B	**54.8**	45.2
	33B	**60.3**	39.7

Table 4: Win rates (%) for **SC** vs. **MC** generation. We find a bias in LLMs towards generating **SC** over **MC**.

code demonstrations. All results are the average of five independent runs with different random seeds.

In Table 1, we observe, as previously reported, that the performance of **TMC** is slightly better than **TSC**.[9] However, their marginal performance gaps raise questions about the impact of modularity.

In Table 2, the relationship between modularity and performance becomes less clear. When comparing **MC** to **SC**, we observe that **MC** consistently underperforms **SC**, which contradicts previous findings. Furthermore, the comparison between **TMC** and **TSC**—a more controlled setting for evaluating modularity—shows no clear correlation between code modularity and performance. This is despite the fact that the transformation process by GPT-3.5-Turbo (**MC** $\rightarrow$ **TMC**, **SC** $\rightarrow$ **TSC**) seems to contribute to non-trivial increases in performance. Therefore, we argue that the previously reported effectiveness of modularity on performance was likely due to unforeseen consequences of the transformation process, rather than the modularity itself.

6 Analysis

6.1 Correlation Study

We conduct an extra experiment to dive deeper into the modularity-performance relationship. Specifically, given 100 code samples used as demonstrations,[10] we compute the Pearson and Spearman correlations between their modularity (MoS) and resulting performance (pass@1). For simplicity, we perform one-shot ICL on CodeContests. Experimental results are presented in Table 3 and Figure 6 in Appendix. Surprisingly, the results reveal weak negative correlations between modularity and performance, suggesting that modularity may not offer benefits, or even hinder performance in some cases.

6.2 Do LLMs Prefer Modular Code?

The minimal performance gap between **(T)MC** and **(T)SC** suggests that LLMs may not have a strong preference for generating modular code. To verify this hypothesis, we experiment to compare the likelihood of LLMs generating modular versus non-modular solutions for the same problem. Formally, the normalized probability of generating a code snippet $\mathcal{C}$ given a problem description $\mathcal{D}$ is:

$$P(\mathcal{C}|\mathcal{D}) = \frac{1}{n}\prod_{t=0}^{n-1} P(x_{t+1}\,|\,\mathcal{D}, x_{\leq t}),$$

where $\mathcal{C}$, consisting of tokens $x_1, \ldots, x_n$, belongs to either **MC** ($\mathcal{C}_{\mathbf{MC}}$) or **SC** ($\mathcal{C}_{\mathbf{SC}}$). We sample nearly 9,000 problems from CodeContests containing both $\mathcal{C}_{\mathbf{MC}}$ and $\mathcal{C}_{\mathbf{SC}}$. We then compare $P(\mathcal{C}_{\mathbf{MC}}|\mathcal{D})$ and $P(\mathcal{C}_{\mathbf{SC}}|\mathcal{D})$ to identify which kind of code is preferred more frequently by LLMs.

Table 4 supports our theory, highlighting a preference for **SC** by LLMs over **MC**. Our findings align with Le et al. (2024) and Li et al. (2024), who observed a performance drop when using (naïvely defined) **MC** demonstrations for encouraging modular code generation. The analysis implies limitations in LLMs' ability to consistently generate well-modularized code, although they might have been exposed to such code during training.

7 Conclusion

In this work, we propose a metric, called MoS, for quantifying the modularity of code snippets and evaluate its impact on performance. Our evaluation reveals no significant correlation, or even a possible weak negative correlation, between modularity and performance. This suggests that factors influencing the usefulness of code examples may differ between human and LLM perspectives. Exploring the influence of other code properties beyond modularity is a promising direction for future work.

[9] For APPS, we conducted experiments only with **TMC** and **TSC** due to computational constraints.

[10] For balanced sampling, we create bins along the MoS dimension and sample an equal number of data from each bin. All the examples are either **MC** or **SC** type.

Limitations

Due to limited computational resources, we focused on designing focused yet generalizable experimental settings. This limited the scope of our investigation, but considering more extensive configurations, such as fine-tuning, employing much larger models, and evaluating other programming languages, in future work will help validate and potentially broaden the applicability of our findings. Despite these limitations, we believe our findings offer valuable insights due to our comprehensive exploration of the feasible configurations within the available resources. Furthermore, identifying a core factor besides modularity that directly affects performance holds significant promise for improving code generation.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Gemini Team. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. 2024. LLM-assisted code cleaning for training accurate code generators. In *The Twelfth International Conference on Learning Representations*.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *Preprint*, arXiv:2310.08992.

Jingyao Li, Pengguang Chen, and Jiaya Jia. 2024. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *Preprint*, arXiv:2312.15960.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

T.J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Appendix

Dataset Filtering. In both the APPS and CodeContests datasets, there are some solution codes that are incorrect based on functional correctness. We filter out the code snippets which cannot pass test-cases from the dataset. After data filtering, APPS has a training dataset of approximately 2K samples, while CodeContest has a training dataset of around 8K samples. Additionally, we also guarantee that both **TMC** and **TSC** pass the test cases. Since some of the problems in APPS provide insufficient or absent test cases, we retain only problems obtained from atcoder, codechef, and codeforces in APPS, following Jain et al. (2024).

Details on Two-shot In-Context Learning. Following Rozière et al. (2024), we use a special instruction to help models understand the specific question format: "*read from and write to standard IO*" for standard questions and "*use the provided function signature*" for call-based questions, which we insert into our prompt as the question guidance for APPS and use special instructions for standard questions for CodeContests. This corresponds to {FEW_SHOT_QUESTION} in Figure 3.

Scatter Plots for Correlation Study. Each data point in the plots indicates code generation performed using one-shot ICL. To conduct experiments using codes with various modularity (MoS), we use codes from the CodeContests dataset. Specifically, we utilize a candidate pool of approximately 8K filtered codes to sample 100 codes. It is important to note that the MoS scores in the demonstration exhibit a wide distribution as depicted in Figure 6.

```python
def check(vec):
    if len(vec) == 0:
        return True
    h = 0
    for v in sorted(vec, reverse=True):
        if h + v[0] < 0:
            return False
        h += v[1]
    return True


def parse(brackets):
    m = 0
    f = 0
    for c in brackets:
        if c == "(":
            f += 1
        else:
            f -= 1
        if m > f:
            m = f
    return m, f


def main():
    N = int(input())
    ls = []
    rs = []
    tot = 0
    for _ in range(N):
        m, f = parse(input().strip())
        tot += f
        if f >= 0:
            ls.append([m, f])
        else:
            rs.append([m-f, -f])
    return tot == 0 and check(ls) and check(rs)


if __name__ == "__main__":
    print("Yes" if main() else "No")
```

(a) **MC** (MoS = 1)

```python
n,m=map(int,input().split())
i=n+1
for i in range(i,m+1):
    t=0
    for j in range(2,i):
        if(i%j==0):
            t=1
            break

    if((i==m and t==1 )or t==0 and i!=m):
        print("NO")
        break
    elif(i==m and t==0):
        print("YES")
        break
    else:
        continue
```

(b) **SC** (MoS = 0)

```python
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def find_next_prime(n):
    i = n + 1
    while True:
        if is_prime(i):
            return i
        i += 1

def check_for_black_day(n, m):
    next_prime = find_next_prime(n)
    if next_prime == m:
        return "YES"
    else:
        return "NO"

def main():
    n, m = map(int, input().split())
    result = check_for_black_day(n, m)
    print(result)

if __name__ == '__main__':
    main()
```

(c) **TMC** (MoS = 1)

```python
n, m = map(int, input().split())
found = False
num = n + 1

while True:
    if num < 2:
        prime = False
    else:
        prime = True

        for i in range(2, int(num**0.5) + 1):
            if num % i == 0:
                prime = False
                break

    if prime:
        if num == m:
            print("YES")
            found = True
        break
    num += 1

if not found:
    print("NO")
```

(d) **TSC** (MoS = 0)

Figure 2: Examples of four code categories corresponding to the same problem.

```
QUESTION:
{PROBLEM}

ANSWER:
```python
{SOLUTION}
```
Refactor the above program. Follow the guidelines
* make the program more modular with smaller and meaningful helpfer functions
* good descriptive names for the helper functions
* have an entry function called ‘main()’
* ‘ main ( )‘ is called inside ‘if __name__ == ‘__main__’’

Do not change the original semantics of the program significantly and no need
to perform optimizations. Enclose the program within backticks as shown above.
```

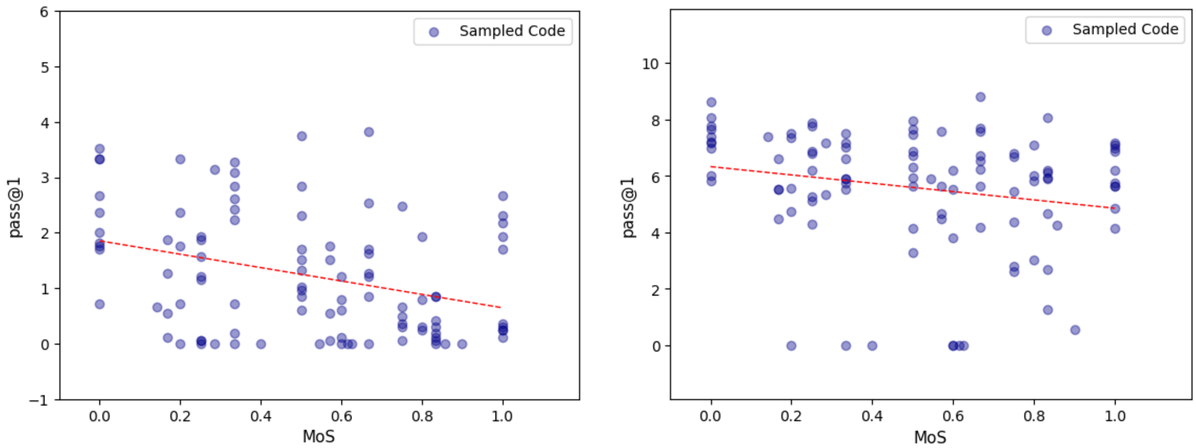
Figure 3: Prompt template for converting **SC** to **TMC**.

```
Q: Write a python code to solve the following coding problem that obeys the
constraints and passes the example test cases. The output code needs to
{FEW_SHOT_QUESTION_GUIDE}. Please wrap your code answer using ```:
{FEW_SHOT_PROMPT}
A: ```{FEW_SHOT_ANSWER}```
Q: Write a python code to solve the following coding problem that obeys the
constraints and passes the example test cases. The output code needs to
{FEW_SHOT_QUESTION_GUIDE}. Please wrap your code answer using ```:
{FEW_SHOT_PROMPT}
A: ```{FEW_SHOT_ANSWER}```
Q: Write a python code to solve the following coding problem that obeys the
constraints and passes the example test cases. The output code needs to
{QUESTION_GUIDE}. Please wrap your code answer using ```:
{PROMPT}
A:
```

Figure 4: Prompt template for two-shot in-context learning with Code Llama.

```
Q: Write a python code to solve the following coding problem that obeys the
constraints and passes the example test cases. The output code needs to
{FEW_SHOT_QUESTION_GUIDE}. Please wrap your code answer using ```:
### Instruction:
{FEW_SHOT_PROMPT}
### Response:
```{FEW_SHOT_ANSWER}```
### Instruction:
{FEW_SHOT_PROMPT}
### Response:
```{FEW_SHOT_ANSWER}```
### Instruction:
{PROMPT}
### Response:
```

Figure 5: Prompt template for two-shot in-context learning with DeepSeekCoder.



(a) One-shot ICL with CodeLlama 7B.

(b) One-shot ICL with DeepSeekCoder 6.7B.

Figure 6: Scatter plots with modularity (MoS) on the x-axis and performance (pass@1) on the y-axis show weak negative correlations between the two variables. The CodeContests dataset is used for evaluation.