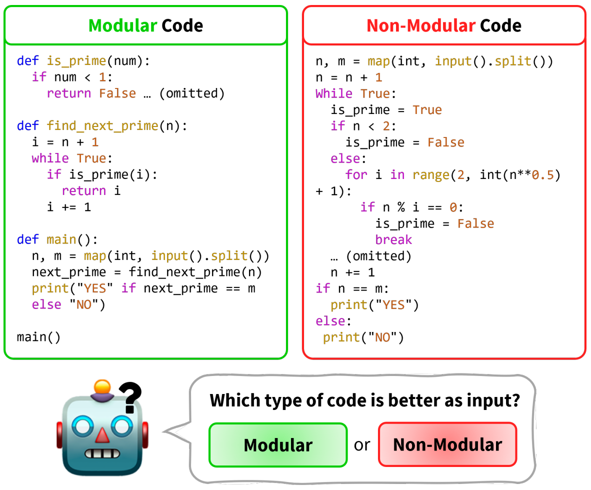
Revisiting the Impact of Pursuing Modularity for Code Generation

Deokyeong Kang[†], Ki Jung Seo[†], Taeuk Kim[*]
Department of Computer Science, Hanyang University, Seoul, Republic of Korea
`{rkdejrdud88,tjrlwjd1,kimtaeuk}@hanyang.ac.kr`

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. Our code is available at https://github.com/HYU-NLP/Revisiting-Modularity.



```
def is_prime(num):
  if num < 1:
    return False … (omitted)

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

def main():
  n, m = map(int, input().split())
  next_prime = find_next_prime(n)
  print("YES" if next_prime == m
  else "NO")

main()
```

```
n, m = map(int, input().split())
n = n + 1
While True:
  is_prime = True
  if n < 2:
    is_prime = False
  else:
    for i in range(2, int(n**0.5)
+ 1):
      if n % i == 0:
        is_prime = False
        break
  … (omitted)
  n += 1
if n == m:
  print("YES")
else:
 print("NO")
```

Figure 1: In this work, we address the following research question: Given modular and non-modular code snippets with identical functionality, which code type more effectively enhances performance in code generation when used as input for code language models?

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. 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. The diversity of code formats also poses challenges in fine-tuning setups, as constructing an appropriate training dataset becomes non-trivial. It is thus important to understand what characteristics of the code provided to the agents contribute to their 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

[†]Equal contribution. [*]Corresponding author.

(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 notorious for their verbosity, it is unclear whether the conversion process aimed solely for modularity or accidentally introduced unexpected side effects. Second, the lack of a formally defined quantitative method for estimating modularity hinders more extensive analyses related to the problem.

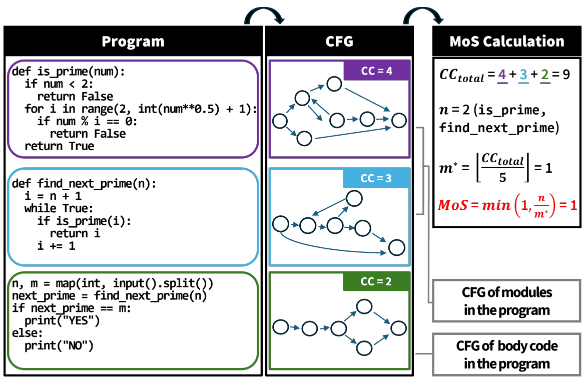
In this paper, we (re-)investigate the effectiveness of pursuing modularity in NL2Code generation. We aim to push the boundaries of previous work by (1) introducing a novel metric that quantifies the modularity of a code snippet using numeric values. Based on the metric, we (2) classify code snippets 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) to investigate the impact of model scale. Figure 1 illustrates the core research question of this work.

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 one.

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 the concept of **Cyclomatic Complexity (CC)** (McCabe, 1976) to determine the ideal number of modules, m^*, for a given code snippet. CC counts the number of independent execution


Program
CFG
MoS Calculation
CC = 4
CC = 3
CC = 2
$CC_{total} = 4 + 3 + 2 = 9$
$n = 2$ (is_prime, find_next_prime)
$m^* = \left\lfloor \frac{CC_{total}}{5} \right\rfloor = 1$
$MoS = min\left(1, \frac{n}{m^*}\right) = 1$
CFG of modules in the program
CFG of body code in the program

```
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
```

```
n, m = map(int, input().split())
next_prime = find_next_prime(n)
if next_prime == m:
  print("YES")
else:
  print("NO")
```

Figure 2: Procedure of computing Cyclomatic Complexity (CC) and Modularity Score (). We first build control-flow graphs (CFGs) from the given code to derive CC. The CC values are then used to compute MoS as the form of CC_{total} and m^*.

paths in the control-flow graph (CFG) of the target code, where the CFG is a graph representation of all potential paths that a program might follow during execution. CC can also be calculated as E - N + 2, where E and N correspond to the number of edges and nodes in the CFG.[4] The CC values are 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.[5] 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:

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

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

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

[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.

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

[4] In practice, we rely on the Python library Radon (https://radon.readthedocs.io/) to derive CC.

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

where n is equal to the actual number of modules in the target code.[6] That is, the closer n (actual number of modules) is to m^* (ideal number of modules), the higher the modularity is considered to be.[7] The process of deriving MoS is illustrated in Figure 2.

In Appendix A, we show that MoS is effective not only in capturing the structural properties of a code snippet but also in revealing the frequency of interactions between functions within the code.

3 Four Code Categories by Modularity

With a way to quantify code modularity, we can now classify a code dataset into two categories—modular and non-modular (= singular). We further leverage prior research by including LLM-based code transformations and their corresponding manually recovered counterparts for controlled experiments. This allows us to create four distinct clusters of code separated by their modularity levels.[8]

Modular Code (MC) is a collection of code snippets with high 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 while preserving its original functionality. 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**.[9]

Transformed Singular Code (TSC) is a variation from **TMC**, whose modularity is manually removed by human programmers. The goal of this approach is to ensure that all factors except modularity are preserved during the conversion process from **TMC** to **TSC**, minimizing unintended changes that could occur if we rely solely on automatic conversion.

Specifically, **TSC** is created by replacing the module invocation parts in **TMC** with the body of the corresponding modules and then removing those modules from the program. By comparing **TSC** and **TMC**, which are expected to be identical except for their modularity, we gain a valuable opportunity to rigorously assess the impact of modularity while accounting for the influence of the transformation process executed by f.

4 Experimental Setups

We explore the impact of modularity by comparing how the four code collections, categorized by their modularity levels, affect performance. We first 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.[10] In addition, we explore the scenario of fine-tuning LLMs with datasets that have varying levels of modularity.

Models We exploit three LLMs for code generation—Code Llama (7B, 34B; Rozière et al., 2024), DeepSeekCoder (6.7B, 33B; Guo et al., 2024), and GPT-4o-mini.[11]

Datasets We employ two NL2Code generation datasets—APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022).[12] They are based on competitive programming contests and provide a set of different solutions for each problem.[13] In this study, we focus our evaluation on Python.

For ICL experiments, **MC** and **SC** demonstrations are chosen from solutions for random problems sampled from each dataset. The one with the highest MoS among solutions is chosen as **MC**. After selecting **SC** examples, they are converted into **TMC**, and finally, **TSC** is manually derived.

For fine-tuning, we split the original dataset into two subsets, **MC** and **SC**, and train different variations of LLMs on each subset. The details on fine-tuning experiments are presented in Appendix C.

[6]We consider modules valid only if they are utilized in at least one execution path of the program.

[7]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 ($n > 0$).

[8]Figure 3 in Appendix displays examples of each category.

[9]See Figure 4 for prompt details on the conversion process.

[10]Refer to Figure 5, 6, and 7 for prompt details.

[11]`https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`

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

[13]We preprocess the APPS and CodeContests datasets following Jain et al. (2024). Refer to Appendix B for more details.

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	**MC**	7.98	12.75	1.26	2.63	0.00	0.03	2.43	4.33
		SC	11.12	15.78	1.65	3.13	0.07	0.26	3.32	5.29
		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	**MC**	24.76	32.59	6.49	10.58	0.72	1.72	9.39	14.01
		SC	28.93	36.26	7.17	11.02	0.65	1.42	10.74	14.99
		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 for generating code given natural language queries.

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
GPT-4o-mini	-	**MC**	14.07	-
		SC	**15.35**	-
		TMC	14.29	-
		TSC	14.4	-

Table 2: Results on CodeContests measured by pass@k. We use n = 10 for pass@1 and n = 50 for pass@10, respectively. The best results are in **bold** for each section. Two-shot prompting is applied for generating code given natural language queries. Due to the cost issue, we only compute pass@1 for GPT-4o-mini.

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 the results of experiments conducted in the ICL setting on APPS and CodeContests, categorized by the modularity of the code demonstrations. All results are the average of five independent runs with different random seeds.

In Table 1, we observe that that **SC** outperforms **MC**, but as previously reported, the performance of **TMC** is slightly better than **TSC**. 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 (**SC** $\rightarrow$ **TMC**) seems to contribute to non-trivial increases in performance, particularly for Code Llama and DeepSeekCoder. GPT-4o-mini demonstrates consistent performance across all four code types, suggesting that modularity does not significantly impact its performance.

We thus 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.

On the other hand, the performance of LLMs fine-tuned on **MC** and **SC** from CodeContests is reported in Table 3. We discover that **SC** constantly

Model	Size	Code Type	CodeContests	
			pass@1	pass@10
Code Llama	7B	**MC**	3.88	12.2
		SC	**4.42**	**12.56**
DeepSeekCoder	6.7B	**MC**	6.06	13.82
		SC	**8.73**	**16.16**

Table 3: Performance of fine-tuning code LLMs on CodeContests, measured by pass@k. We use n = 10 for pass@1 and n = 50 for pass@10. The best results in each section are highlighted in **bold**. Zero-shot prompting is used during inference, meaning no demonstrations are provided to guide the models in generating code.

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

outperforms **MC**, albeit by a narrow margin, reflecting a simlar trend observed in the ICL setting.[14] These results suggest that the modularity of the code examples used for training does not have a significant impact on the performance of LLMs in terms of code generation.

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,[15] 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 4 and Figure 8 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.

[14]Due to the cost of constructing **TMC** and **TSC** using GPT-3.5-Turbo, we focused our experiments on **MC** and **SC**.

[15]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.

Model	Size	$PPL(\mathcal{C}_{\text{MC}})$	$PPL(\mathcal{C}_{\text{SC}})$
Code Llama	7B	2.2 (0.57)	2.4 (1)
	34B	2.02 (0.45)	2 (0.44)
DeepSeekCoder	6.7B	1.93 (0.41)	2.05 (0.63)
	33B	1.89 (0.42)	1.89 (0.42)

Table 5: Perplexities of LLMs for $\mathcal{C}_{\text{MC}}$ and $\mathcal{C}_{\text{SC}}$. LLMs exhibit similar predictive ability for both **SC** and **MC**. Numbers in parentheses represent standard deviations.

6.2 Do LLMs Favor 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 compare the perplexities of LLMs on modular and non-modular code. Formally, the perplexity of a code snippet $\mathcal{C}$ given a problem description $\mathcal{D}$ is:

$$PPL(\mathcal{C}) = exp\left\{-\frac{1}{n}\sum_{t=0}^{n-1} logP(x_{t+1}\,|\,\mathcal{D}, x_{\leq t})\right\},$$

where $\mathcal{C}$, consisting of tokens $x_1, \ldots, x_n$, belongs to either **MC** ($\mathcal{C}_{\text{MC}}$) or **SC** ($\mathcal{C}_{\text{SC}}$). We sample nearly 10,000 problems from CodeContests containing both $\mathcal{C}_{\text{MC}}$ and $\mathcal{C}_{\text{SC}}$, with $\mathcal{C}_{\text{MC}}$ having MoS values ranging from 0.7 to 1 and $\mathcal{C}_{\text{SC}}$ having value of 0. We then compare $PPL(\mathcal{C}_{\text{MC}})$ and $PPL(\mathcal{C}_{\text{SC}})$ averaged over all examples to identify which type of code is better predicted by code language models.

Table 5 supports our hypothesis, highlighting a neutral preference of LLMs which is not biased towards generating **SC** or **MC**. This is presumably because the models were likely exposed to both code types during pre-training. We speculate that this could be one of the reasons why modular examples are not always beneficial for code generation in language models.

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.

Limitations

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

Ethics Statement

In this study, we utilize models and datasets publicly available on Huggingface, ensuring that no ethical issues are associated with their usage. All datasets for evaluation are open-source and follow strictly to data usage policies.

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2020-II201373, Artificial Intelligence Graduate School Program (Hanyang University)), Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2024-RS-2023-00253914) grant funded by the Korea government (MSIT), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2018R1A5A7059549).

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*.

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.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. *Preprint*, arXiv:1711.05101.

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

A The Effectiveness of MoS

In this section, we explore the question: Does MoS indeed reflect the modularity of code? A key aspect of modular programming is decomposing a program into modules and enabling interaction between them through function calls or class instantiation. We argue that MoS not only considers the structure of a program but also inherently reflects the interactions between its constituents (e.g., functions and class methods).

To support this claim, we examine the correlation between MoS and the number of function calls to determine if MoS reflects module interactions. Specifically, we (1) sample 100 codes from the CodeContests training set, (2) calculate the Pearson and Spearman correlation coefficients between each code example's MoS and the frequency of the function invocations, and (3) repeat the same experiment using five random seeds. To ensure balanced sampling, we create bins based on MoS values and sample the same number of codes from each bin. As shown in Table 6, the Pearson and Spearman correlation coefficients are 0.41 and 0.61, respectively. This positive correlation between MoS and the frequency of function calls highlights its effectiveness in reflecting modular interaction.

Random Seed	Pearson	Spearman
27	0.37 (0)	0.61 (0)
42	0.41 (0)	0.64 (0)
101	0.4 (0)	0.63 (0)
134	0.44 (0)	0.57 (0)
169	0.42 (0)	0.62 (0)
Average	**0.41**	**0.61**

Table 6: Correlations between MoS of code and number of function calls in the code. Numbers in parentheses represent p-values.

B Dataset Preprocessing

In both the APPS and CodeContests datasets, there are some solution codes that are incorrect based on functional correctness. We filter out code snippets that do not pass the test cases and retain only the solutions written in Python. After data filtering, CodeContest has a training dataset of around 7K samples, while APPS has a training dataset of approximately 2K samples. 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). APPS are divided into APPS-INTRODUCTORY, APPS-INTERVIEW, and APPS-COMPETITION based on problem difficulty. Table 7 describes the statistics of the APPS and CodeContests datasets we finally employed. Additionally, we also guarantee that both **TMC** and **TSC** pass the test cases.

C Training Details

We construct two training datasets consisting solely of **MC** and **SC**, respectively, based on their MoS values, and fine-tune the full parameters of the code LLMs on these datasets. Both training subsets cover the same set of problems, with **SC** having a MoS value of 0 and **MC** having MoS values ranging between 0.7 and 1. Following Jain et al. (2024), we employ minhash-based deduplication using Gaoya[16] to limit each problem to a maximum of 25 solutions. Then, an equal number of codes are randomly selected from each problem to ensure that the **MC** and **SC** datasets have the same number of training samples. Applying this process to the CodeContests dataset results in about 5K problems and 61K training examples for both the **MC** and **SC** datasets. We decided to exclude the APPS dataset from the fine-tuning experiment due to the limited number of common problems and codes for **SC** and **MC**.

For model training, we used the HuggingFace Trainer[17] library with the AdamW(Loshchilov and Hutter, 2019) optimizer, starting with a learning rate of 5e-5. A cosine learning rate scheduler with a warmup ratio of 0.01 was applied, and we utilized bf16 precision to optimize memory usage. The effective batch size was set to 64, achieved through a per-device batch size of 4 and gradient accumulation step of 16. Training was conducted for 1 epoch on 4 A6000 GPUs. After training, model inference was conducted in a zero-shot manner using the same sampling parameters as those in the ICL setting.

[16]https://github.com/serega/gaoya
[17]https://huggingface.co/docs/transformers/main_classes/trainer

D Details on 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 4. We use temperature and top-p sampling strategies for calculating pass@k. Following Jain et al. (2024), we set the top-p value of 0.95 and temperature to 0.1 for pass@1, and 0.6 for pass@10.

```python
import sys


def check_bombs(map, pos, dir, range=3):
    if dir == 'up':
        x, y = pos
        while range > 0 and y > 0:
            if map[y-1][x] == '1':
                return (x, y-1)
            else:
                y -= 1
                range -= 1
    elif dir == 'down':
        x, y = pos
        while range > 0 and y < len(map)-1:
            if map[y+1][x] == '1':
                return (x, y+1)
            else:
                y += 1
                range -= 1
    elif dir == 'left':
        x, y = pos
        while range > 0 and x > 0:
            if map[y][x-1] == '1':
                return (x-1, y)
            else:
                x -= 1
                range -= 1
    elif dir == 'right':
        x, y = pos
        while range > 0 and x < len(map[0])-1:
            if map[y][x+1] == '1':
                return (x+1, y)
            else:
                x += 1
                range -= 1
    else:
        return None


def chain_bombs(map, init_pos):
    lmap = map[:]
    detonated = [init_pos]
    while detonated:
        x, y = detonated.pop()
        lmap[y][x] = '0'
        res = check_bombs(lmap, (x, y), 'up')
        if res:
            detonated.append(res)
        res = check_bombs(lmap, (x, y), 'down')
        if res:
            detonated.append(res)
        res = check_bombs(lmap, (x, y), 'left')
        if res:
            detonated.append(res)
        res = check_bombs(lmap, (x, y), 'right')
        if res:
            detonated.append(res)
    return lmap


def main(args):
    data_set = int(input())
    maps = []
    init_pos = []
    for i in range(data_set):
        _ = input()
        maps.append([list(input().strip()) for _ in range(8)])
        init_pos.append([int(input())-1, int(input())-1])

    count = 1
    for map, pos in zip(maps, init_pos):
        result = chain_bombs(map, pos)
        print('Data {}:'.format(count))
        for row in result:
            print(''.join(row))
        count += 1


if __name__ == '__main__':
    main(sys.argv[1:])
```

MC (MoS = 0.5)

```python
n = int(input())
for i in range(n):
    print("Data ", i+1, ":", sep='')
    input()
    a = [[0 for c in range(15)] for r in range(15)]
    for r in range(8): a[r] = list(input())
    Q = []
    c,r = int(input())-1, int(input())-1
    Q.append((r, c))
    while len(Q):
        r, c = Q.pop(0)
        a[r][c] = '0';
        for nr in range(r-3, r+4):
            if nr >= 0 and nr < 8:
                if a[nr][c] == '1':
                    Q.append((nr, c))
                    a[nr][c] = '2'
        for nc in range(c-3, c+4):
            if nc >= 0 and nc < 8:
                if a[r][nc] == '1':
                    Q.append((r, nc))
                    a[r][nc] = '2'
    for r in range(8):
        t = ''.join(a[r])
        s = t.replace("2", "0")
        print(s[0:8], sep='')
```

SC (MoS = 0)

```python
def initialize_grid():
    return [['0' for _ in range(8)] for _ in range(8)]

def read_grid():
    return [list(input().strip()) for _ in range(8)]

def read_initial_bomb_coordinates():
    col = int(input().strip()) - 1
    row = int(input().strip()) - 1
    return row, col

def process_bomb_queue(queue, grid):
    directions = [(-3, 0), (-2, 0), (-1, 0), (1, 0), (2, 0), (3, 0),
                  (0, -3), (0, -2), (0, -1), (0, 1), (0, 2), (0, 3)]
    while queue:
        r, c = queue.pop(0)
        grid[r][c] = '0'
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < 8 and 0 <= nc < 8 and grid[nr][nc] == '1':
                queue.append((nr, nc))
                grid[nr][nc] = '2'

def print_final_grid(grid, dataset_index):
    print(f"Data {dataset_index}:")
    for row in grid:
        print("".join(row).replace('2', '0'))

def main():
    n = int(input())
    for dataset_index in range(1, n + 1):
        input()  # skip the blank line
        grid = read_grid()
        initial_bomb_row, initial_bomb_col = read_initial_bomb_coordinates()
        queue = [(initial_bomb_row, initial_bomb_col)]
        process_bomb_queue(queue, grid)
        print_final_grid(grid, dataset_index)

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

TMC (MoS = 1)

```python
n = int(input())
for dataset_index in range(1, n + 1):
    input()  # skip the blank line
    grid = [list(input().strip()) for _ in range(8)]
    initial_bomb_col = int(input().strip()) - 1
    initial_bomb_row = int(input().strip()) - 1

    queue = [(initial_bomb_row, initial_bomb_col)]
    directions = [(-3, 0), (-2, 0), (-1, 0), (1, 0), (2, 0), (3, 0),
                  (0, -3), (0, -2), (0, -1), (0, 1), (0, 2), (0, 3)]
    while queue:
        r, c = queue.pop(0)
        grid[r][c] = '0'
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < 8 and 0 <= nc < 8 and grid[nr][nc] == '1':
                queue.append((nr, nc))
                grid[nr][nc] = '2'

    print(f"Data {dataset_index}:")
    for row in grid:
        print("".join(row).replace('2', '0'))
```

TSC (MoS = 0)

Figure 3: Examples of four code categories for the same problem with their modularity scores.

	Split	CodeContests	APPS (INTRODUCTORY)	APPS (INTERVIEW)	APPS (COMPETITION)
# Problems	Training	7313	42	1247	361
	Test	165	702	2699	309
# Avg. Test Cases	Training	20	1	1	10
	Test	10	16	24	45
# Avg. Solutions	Training	182	64	24	17

Table 7: Statistical details regarding the number of problems, the average number of test cases per problem, and the average number of solutions in the filtered datasets of CodeContests and APPS.

~~~~
QUESTION:
{PROBLEM}

ANSWER:
```python
{SOLUTION}
```
Refactor the above program. Follow the guidelines
* make the program more modular with smaller and meaningful helper 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 4: The prompt template used 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:
~~~~

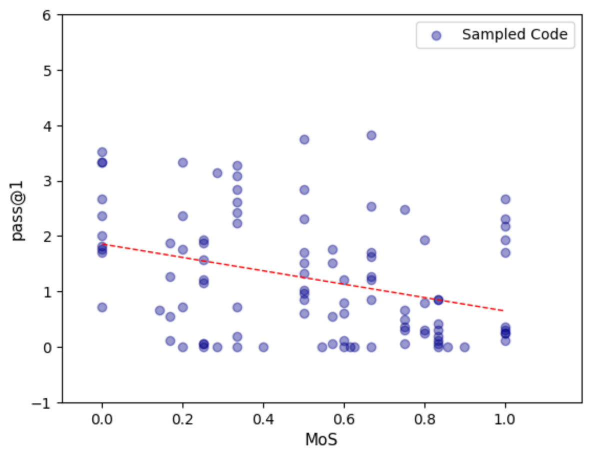
Figure 5: The prompt template used for two-shot in-context learning with Code Llama.

~~~
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:
~~~

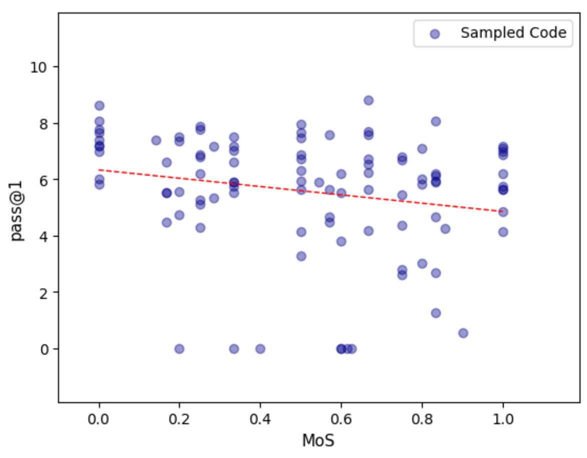
Figure 6: The prompt template used for two-shot in-context learning with DeepSeekCoder.

```
SYSTEM: You are an AI programming assistant.
USER: Write a python code to solve the following coding problem that obeys
the constraints and passes the example test cases.
USER: {FEW_SHOT_PROMPT}
ASSISTANT: {FEW_SHOT_ANSWER}
USER: {FEW_SHOT_PROMPT}
ASSISTANT: {FEW_SHOT_ANSWER}
USER: {PROMPT}
ASSISTANT:
```

Figure 7: The prompt template used for two-shot in-context learning with GPT-4o-mini.



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



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

Figure 8: 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. Note that the MoS scores of demonstration codes exhibit a wide distribution. The red dashed line represents the regression line.