# How Natural Language Proficiency Shapes GenAI Code for Software Engineering Tasks

Ruksit Rojpaisarnkit*, Youmei Fan*, Kenichi Matsumoto* and Raula Gaikovina Kula†
*Nara Institute of Science and Technology, Japan, †The University of Osaka, Japan,
rojpaisarnkit.ruksit.rn1@is.naist.jp, fan.youmei.fs2@is.naist.jp, matumoto@is.naist.jp,
raula-k@ist.osaka-u.ac.jp

***Abstract*—With the widespread adoption of Foundation Model (FM)-powered tools in software engineering, the natural language prompt has become a critical interface between developers and Large Language Models (LLMs). While much research has focused on prompt structure, the natural language proficiency is an underexplored factor that can influence the quality of generated code. This paper investigates whether the English language proficiency itself independent of the prompting technique affects the proficiency and correctness of code generated by LLMs. Using the HumanEval dataset, we systematically varied the English proficiency of prompts from basic to advanced for 164 programming tasks and measured the resulting code proficiency and correctness. Our findings show that LLMs default to an intermediate (B2) natural language level. While the effect on the resulting code proficiency was model-dependent, we found that higher-proficiency prompts consistently yielded more correct code across all models. These results demonstrate that natural language proficiency is a key lever for controlling code generation, helping developers tailor AI output and improve the reliability of solutions.**

***Keywords*-Code proficiency, LLMs, CEFR**

## I. Introduction

Foundation Model (FM)-powered coding assistants are disrupting the software development landscape. Developers are utilizing AI technologies like GitHub Copilot[1], ChatGPT[2], and Gemini[3] to generate code and solve software engineering tasks. In recent years, Large Language Models (LLMs) have become powerful coding assistants, particularly for code generation [1], [2]. The efficacy of these models heavily relies on the natural language prompts they receive, making prompt quality a key determinant of output quality [3].

However, while much research has focused on prompt engineering and model fine-tuning to improve results [4], [5], less attention has been paid to how the natural language complexity of the prompt itself affects the generated code. The proficiency of the generated code is a critical, yet overlooked factor. A mismatch between the code complexity and the user's skill level can create significant issues. This can lead to novices being overwhelmed by overly complex code or experts being frustrated by simplistic, inefficient solutions. Consider a developer who is highly proficient at describing tasks in natural language (e.g., C1-level proficiency) but has only a novice-level understanding of the programming language (e.g., A1-level proficiency). If a LLMs respond to the sophisticated prompt and generates code that is also at a C1-level proficiency, this developer may face significant challenges in understanding, debugging, and safely integrating the code. Recent studies show that students and practitioners often struggle to understand LLM-generated code due to unfamiliar code styles, an overestimation of its correctness, and their own limited programming proficiency [6]. Furthermore, other research has found that beginning programmers struggle to write effective prompts and often misunderstand the resulting code, revealing a fundamental disconnect between novice users and LLMs [7].

This challenge extends to various settings. In a collaborative industry environment, it can create significant friction and increase review overhead. For instance, if a senior developer uses LLMs to produce highly complex, idiomatic code, junior team members may struggle to understand, review, and later maintain it, impacting team velocity. Furthermore, modern software teams often operate in environments where developers have varied expertise across multiple languages. Ensuring AI-generated code aligns with their specific proficiency in the target language is crucial for preventing subtle bugs [8]. Therefore, tailoring the proficiency of generated code to user expertise is a critical factor for the successful adoption of LLMs.

[1]https://github.com/features/copilot
[2]https://openai.com/index/chatgpt/
[3]https://gemini.google.com/app

| Prompt | Sample AI-Generated Response |
|---|---|
| Translate the following software engineering problem into natural language including function signature, docstring, body, and unit testcases without code. Make sure it fits into one paragraph. | The function 'has_close_elements' takes a list of floating-point numbers, 'numbers', and a floating-point 'threshold'. It checks if there are any two numbers in the list that are closer to each other than the specified threshold. The function returns 'True' if such a pair of numbers exists, and 'False' otherwise. For example, calling 'has_close_elements' with the list '[1.0, 2.0, 3.0]' and a threshold of '0.5' returns 'False' because no two numbers in the list are within '0.5' units of each other. Conversely, calling it with '[1.0, 2.8, 3.0, 4.0, 5.0, 2.0]' and a threshold of '0.3' returns 'True' because at least one pair of numbers is closer than '0.3' units apart. |

TABLE I: AI-Generated Descriptions of Software Engineering Problems

In this paper, we conduct an empirical study using several leading LLMs (GPT-4o, Gemini 2.5 Pro, and Claude Sonnet 4) and the HumanEval dataset [9] to investigate whether the natural language proficiency of a prompt affects the proficiency and correctness of the generated code. Briefly, the proficiency of an artifact (a prompt or generated code) refers to the proficiency level required by a person to author or comprehend it. We modified the English proficiency level of each problem description based on the Common European Framework of Reference for Languages (CEFR), a robust international standard for defining language proficiency.

By prompting we align the CEFR [10] according to the following levels and example words:

- CEFR A1 - Beginner (i.e. about, above)
- CEFR A2 - Elementary (i.e. ability, access, acceptance)
- CEFR B1 - Intermediate (i.e. accidental, accompany)
- CEFR B2 - Upper Intermediate (i.e. absurd, actual)
- CEFR C1 - Advanced (i.e.complexity, reluctantly)
- CEFR C2 - Proficient (i.e. eccentricity, deliberate)

Our study is based on the software engineering tasks dataset from HumanEval, which contains 164 hand-written programming problems in Python. Each problem includes a prompt (doc-string), a canonical solution, and test cases. To address known issues with the original test cases and ensure the reliability of our pass@1 metric, we incorporate the more robust test suite from HumanEvalPlus [9].

To measure code proficiency, we adapt the levels defined by PYCEFR [11], which assigns proficiency ratings to Python elements based on their conceptual difficulty. For example, *print()* and *return* are considered A1 (Beginner), *if/else* and *assert* are B1 (Intermediate), and *zip()* and *map()* are C2 (Proficient).

We then ask the following research questions:

1) **RQ1:** *What is the baseline English proficiency (CEFR level) of problem descriptions generated by LLMs?*
   While the challenges of code comprehension are known, the causal link from natural language prompt proficiency to code proficiency is not well understood. This research question aims to investigate the proficiency level of LLMs-generated descriptions based on the CEFR based proficiency.
2) **RQ2:** *Does the English proficiency (CEFR level) of a natural language prompt influence the code proficiency and correctness of the solutions generated by LLMs?*
   With the baseline established from RQ1, we can then systematically explore the core of the problem. This research question investigates whether the natural language proficiency of problem descriptions influences the code proficiency of solutions generated by LLMs.

Our findings reveal that LLMs tend to generate problem descriptions at a B2 (Upper Intermediate) level or higher, suggesting a baseline English proficiency is required to understand their output. Furthermore, a clear correlation exists for some LLMs: adjusting the natural language proficiency of prompts tend to yield different proficiency levels of code. Accordingly, simplifying the natural language proficiency of the prompt can lower the correctness of the resulting code. These results indicate that the natural language proficiency of a prompt directly influences the reasoning and code generation capabilities of LLMs. The dataset of prompts and responses analyses for each RQ are available in Zenodo, at https://doi.org/10.5281/zenodo.17285168.

## II. CEFR Proficiency of SE Tasks

To answer RQ1, we first prompted the LLMs to transform each original prompt (a docstring from the HumanEval dataset) into a natural language paragraph describing the task, as shown in Table I. This step is to understand the LLM's baseline behavior

| Prompt | Sample AI-Generated Response |
|---|---|
| Increase the english proficiency level of this paragraph to CEFR C2 level | The function 'has_close_elements' is designed to evaluate a list of floating-point numbers, termed 'numbers', along with a specified floating-point 'threshold'. Its purpose is to determine the existence of any two numbers within the list whose proximity is less than the provided threshold. The function will return 'True' if such a pair is identified; otherwise, it will return 'False'. For instance, invoking 'has_close_elements' with the list '[1.0, 2.0, 3.0]' and a threshold of '0.5' results in 'False', as no two numbers in the list differ by less than '0.5' units. In contrast, calling it with '[1.0, 2.8, 3.0, 4.0, 5.0, 2.0]' and a threshold of '0.3' yields 'True', given that at least one pair of numbers is separated by less than '0.3' units. |
| Lower the english proficiency level of this paragraph to CEFR A1 level | The function 'has_close_elements' looks at a list of numbers called 'numbers' and a number called 'threshold'. It checks if any two numbers in the list are closer than the 'threshold'. If yes, it says 'True'. If no, it says 'False'. For example, if we use the list '[1.0, 2.0, 3.0]' and 'threshold' '0.5', it says 'False' because no two numbers are close. But with the list '[1.0, 2.8, 3.0, 4.0, 5.0, 2.0]' and 'threshold' '0.3', it says 'True' because some numbers are close. |
| Show the solution of this problem in source code. Return your response as a JSON object with one fields: 1. "code": the actual Python code. | def has_close_elements(numbers, threshold): for i in range(len(numbers)):for j in range(i + 1, len(numbers)): if abs(numbers[i] - numbers[j]) <threshold:return True return False # Example usage:print(has_close_elements([1.0, 2.0, 3.0], 0.5)) # False print(has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)) # True |

TABLE II: Increase proficiency prompt, Decrease proficiency prompt, and Generate Code prompt

| CEFR Levels | HumanEval Dataset | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *GPT-4o* | | | *Gemini 2.5 Pro* | | | *Claude Sonnet 4* | | |
| | ori. | inc. | dec. | ori. | inc. | dec. | ori. | inc. | dec. |
| C2 | 30 | 79 | 1 | 9 | 115 | 0 | 42 | 143 | 1 |
| C1 | 44 | 67 | 12 | 45 | 38 | 2 | 30 | 19 | 10 |
| B2 | 90 | 18 | 151 | 110 | 11 | 138 | 91 | 2 | 144 |
| B1 | 0 | 0 | 0 | 0 | 0 | 15 | 1 | 0 | 8 |
| A2 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 1 |
| A1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

TABLE III: Distribution of CEFR Levels of 164 SE tasks as Original, Increased and Decreased proficiency problem descriptions (RQ1 and RQ2)

before adjusting the natural language proficiency of the description for our next step.

**CEFR for Natural Language (English).**

To determine the natural language proficiency of these generated descriptions, we utilized a CEFR word list from Kaggle [12], which contains over 10,000 English words mapped to their corresponding CEFR levels (A1 to C2). Our classification method is based on the hypothesis that a text's proficiency is defined by its most advanced vocabulary. Therefore, we assigned an overall proficiency level to each description based on the highest CEFR level of any word it contained. For example, a paragraph containing the B2-level word "abandon" would be classified as B2, even if all other words were A1 or A2.

Table III shows the distribution of the 164 problem descriptions generated by each of the three LLMs. The "ori." columns represent the baseline proficiency levels relevant to RQ1. The results for the original prompts show that the LLMs predominantly generate descriptions at an intermediate proficiency level (CEFR B2) for software engineering tasks. For example, GPT-4o and Claude Sonnet 4 produced 90 and 91 descriptions at the B2 level, respectively. Notably, across all models, descriptions rated below B2 were extremely rare. Only one description (from Claude Sonnet 4) was classified as B1, and none fell into the A1 or A2 beginner levels. This suggests that a user needs at least an intermediate (B2) level of English proficiency to comfortably understand problem descriptions generated by these models.

**RQ1 Summary**

From the 164 problem descriptions, we find that the AI-generated descriptions proficiency is mostly intermediate level(CEFR B2) for SE problem descriptions. Interestingly, there is only one paragraph that contains only very basic words (CEFR A1- CEFR B1) from Claude Sonnet 4.

## III. Prompting Proficiency of the AI-Generated Code

To answer RQ2, for each problem, we prompted the LLMs to rewrite the original description generated in RQ1 into two new versions: one simplified to a basic proficiency level (A1) and another elevated to an advanced level (C2). We then used these *decreased-proficiency* and *increased-proficiency* descriptions as new prompts to generate code solutions. Table II shows examples of these prompts. The overview of this process is depicted in Figure 1.

Our analysis involved two main steps: i.) Validating Prompt Proficiency: We first verified that the LLMs

| Code Proficiency | HumanEval Dataset | | | | | |
|---|---|---|---|---|---|---|
| | *GPT-4o* | | *Gemini 2.5 Pro* | | *Claude Sonnet 4* | |
| | inc. | dec. | inc. | dec. | inc. | dec. |
| C2 | 17 | 17 | 15 | 9 | 7 | 7 |
| C1 | 24 | 25 | 13 | 14 | 14 | 14 |
| B2 | 20 | 15 | 11 | 1 | 10 | 9 |
| B1 | 69 | 41 | 24 | 27 | 20 | 21 |
| A2 | 18 | 26 | 47 | 53 | 63 | 60 |
| A1 | 16 | 40 | 54 | 60 | 50 | 53 |
| **Statistical Test Result** | | | | | | |
| p-value | 0.001** | | 0.057* | | 0.998 | |
| CramerV | 0.244 | | 0.180 | | 0.027 | |
| Effect size | small | | small | | small | |
| **Pass@1 Benchmark Result** | | | | | | |
| HumanEval | 79.3% | 76.2% | 87.2% | 71.3% | 93.9% | 91.5% |
| HumanEval+ | 76.2% | 70.7% | 83.5% | 69.5% | 88.4% | 85.4% |

TABLE IV: Distribution of Code Proficiency in Increased and Decreased proficiency prompt for 164 SE-tasks in HumanEval and Pass@1 result (RQ2).

successfully altered the CEFR levels of the rewritten descriptions to create distinct low-proficiency and high-proficiency prompt sets. ii.) Evaluating Code Proficiency and Correctness: We then measured the proficiency of the resulting code and assessed its correctness using the pass@1 metric against the HumanEval and HumanEvalPlus test suites.

**PYCEFR for Python Code Assessment.** To assess code proficiency, we used the reference levels from PYCEFR [11], a tool that classifies Python code elements into six CEFR-inspired levels (A1-C2). For example, a basic `print()` function corresponds to A1, an `if-else` structure to B1, and an advanced construct like `zip` to C2. We analyzed each generated solution and assigned it an overall proficiency score based on the highest level code element it contained.

### *A. Analysis 1 - Verifying the CEFR proficiency of descriptions*

As shown in Table III, our prompting strategy successfully altered the natural language proficiency of the problem descriptions. For the increasing proficiency task, all models produced a large number of C1 and C2-level descriptions, confirming they could effectively elevate the language. Conversely, for the decreasing proficiency task, models were less effective at simplification. While some descriptions were lowered to B1, A2, or A1, a significant portion remained at the B2 (Intermediate) level, indicating a floor effect in the models' ability to simplify technical language.

### *B. Analysis 2 - Evaluating the impact on the code proficiency of generated solutions*

*Code Proficiency:* Table IV presents the distribution of code proficiency for solutions generated from the increased and decreased-proficiency prompts. We conducted a Chi-Square test of independence to determine if the natural language proficiency level had a statistically significant effect on the code proficiency level.

Our findings vary by model. For GPT-4o, the test yielded a statistically significant result ($p<0.05$), rejecting the null hypothesis and indicating that the natural language proficiency was associated with differences in the distribution code proficiency. For Gemini 2.5 Pro, the result approached significance ($p=0.057$), suggesting a potential but less pronounced relationship. In contrast, Claude Sonnet 4 showed no significant effect ($p=0.998$), implying the generated code from its was independent of the CEFR natural language proficiency. These results suggest that for some LLMs, adjusting natural language proficiency could alter the code proficiency of the generated solution, both increasing and decreasing proficiency.

*Code Correctness:* The impact on code correctness was more uniform across all models. As shown in the Pass@1 Benchmark Result section of Table IV, solutions generated from increased-proficiency prompts consistently achieved higher pass@1 scores on both HumanEval and HumanEvalPlus benchmarks. For every model, the decreased-proficiency prompts led to a noticeable drop in correctness, suggesting that richer, more advanced language enables the models to generate more reliable solutions.

**RQ2 Summary**

Adjusting (increasing or decreasing) the natural language proficiency of the SE task description resulted in the code with different proficiencies for GPT-4o but not Gemini 2.5 Pro and Claude Sonnet 4. However, the generated solution from increased natural language proficiency description tends to achieve higher pass@1 test coverage than the generated solution from decreased natural language proficiency description of three LLMs.

## IV. Limitations

This section outlines the key limitations of our study and potential threats to the validity of our findings.

**Prompt Design.** One limitation stems from the prompt design used to manipulate natural language proficiency. While we aimed to standardize prompts and control for content, subtle variations in phrasing, tone, or complexity may have influenced the quality of the generated responses.

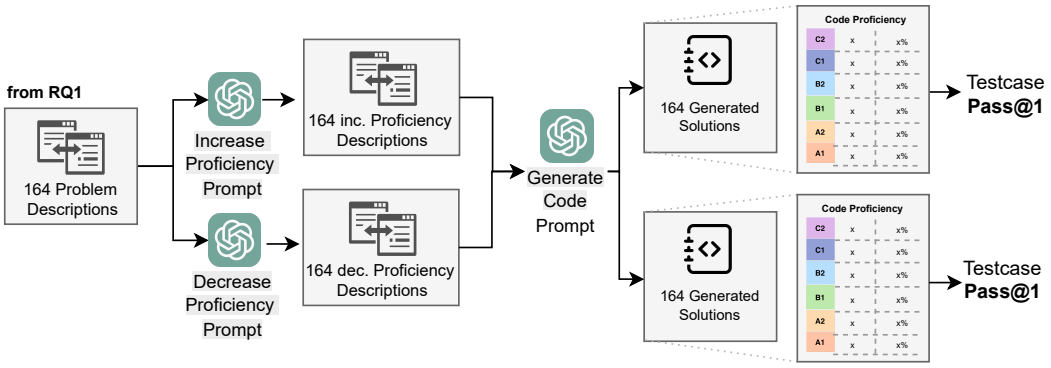



Fig. 1: Evaluating Proficiency level in problem description and Proportion of code proficiency of generated solution when using LLM (Statistic result as shown in Table III and Table IV)

**Proficiency Assessment.** Our approach to assessing both natural language and code proficiency relies on a consistent hypothesis: the presence of advanced elements serves as a strong proxy for overall proficiency. For natural language proficiency, our analysis focuses on the vocabulary within a given text, rather than its grammatical structure or other syntactic aspects. Similarly, for code proficiency, we evaluate the sophistication of the solution based on the specific code constructs and elements it contains. This method provides a focused, rather than holistic, measure of proficiency. However, this approach is grounded in the principle that encountering a single, critical, and unfamiliar element can significantly impede comprehension. For instance, a single unknown yet crucial word can obscure the meaning of an entire paragraph. Likewise, in programming, an unfamiliar function or syntactic structure can prevent a developer from fully understanding the code's intent and functionality. Therefore, the implications of our findings should therefore be understood primarily in the context of construct complexity and its impact on immediate user comprehension. A more comprehensive proficiency model could integrate our construct-based analysis with established software engineering metrics, such as cyclomatic complexity, readability scores, and static analysis for code smells, to provide a richer understanding of AI-generated software quality.

**Task Scope.** Our study is based on the HumanEval benchmark dataset, which focuses on short, self-contained Python programming tasks with a heavy emphasis on algorithmic reasoning. As such, the results may not generalize to broader or more complex domains of software engineering, such as API integration, system design, large-scale code generation, or debugging tasks that involve maintaining context over longer sequences.

**Model Dependency.** All results in this study were obtained using three LLMs (GPT-4o, Gemini 2.5 Pro, and Claude Sonnet 4). While this allows for controlled experimentation, it also limits the generalization of our findings. Other LLMs especially those trained on different data or optimized with different objectives may respond differently to variations in natural language proficiency.

## V. Implications and Future Outlook

**For Developers** Findings from RQ2 demonstrate that the natural language proficiency of prompts could significantly influence the quality and sophistication of code generated by large language models. Developers using AI-assisted code generation tools should therefore be mindful not only of the technical accuracy and completeness of their prompts but also of their natural language clarity and proficiency. Well-structured, high-proficiency language in prompts can lead to the generation of more advanced, correct, and functional code. This insight encourages developers to view prompt engineering as a multi-dimensional practice involving both contextual clarity and language precision. However, this sensitivity is not universal across all models. A development team, particularly one with diverse levels of natural language proficiency (e.g., a mix of native and non-native speakers), might achieve more consistent and predictable results by choosing an LLM that is less sensitive to variations in natural language expression. This ensures that all team members can generate reliable code regardless of their natural language style.

**For Non programmers** The result from RQ1 reveals that large language models can effectively translate structured code prompts into natural language descriptions. This capability opens the door for non-programmers to better understand software

tasks and participate in software-related conversations. However, our results also indicate that these descriptions, even after simplification, tend to remain at a minimum B2 CEFR level. This suggests a potential barrier to accessibility: individuals with lower English proficiency may still struggle to fully grasp software concepts conveyed by large language models.

**For Researchers** This study contributes to the emerging body of research examining the intersection between natural language proficiency and code proficiency. Our findings underscore the significance of natural language factors in prompt-based programming and highlight the measurable influence of language quality on the proficiency of code output. The relationship between natural language and code proficiency, as explored through RQ2, provides a foundation for future research on educational tools, accessibility in programming, and improvements in prompt engineering. Moreover, these results suggest promising directions for designing more inclusive AI systems that adapt to users natural language capabilities while maintaining technical accuracy.

**Future Outlook** It is now clear that AI-assistants and FM-powered tools are here to stay. Therefore, to embrace the technology, we also need to understand how to control not only through the precision of the prompt but also the proficiency and nuances behind the natural language used. We envision that these results lay the groundwork for future research directions into more predictable refining of AI-generated code, assessing developer skills and ultimately the quality of solutions to software engineering tasks.

## VI. Acknowledgement

This work is supported by the JSPS KAKENHI Grant Number JP24H00692 and JP23K28065.

## References

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[2] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can chatgpt support developers? an empirical evaluation of large language models for code generation," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 167–171.

[3] J. T. Liang, C. Yang, and B. A. Myers, "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*, 2024, pp. 1–13.

[4] C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation," *Information and Software Technology*, vol. 175, p. 107523, 2024.

[5] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," *arXiv preprint arXiv:2310.10508*, 2023.

[6] Y. Zi, L. Li, A. Guha, C. Anderson, and M. Q. Feldman, "I would have written my code differently: Beginners struggle to understand llm-generated code," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion 25. ACM, Jun. 2025, pp. 1479–1488. [Online]. Available: http://dx.doi.org/10.1145/3696630.3731663

[7] S. Nguyen, H. M. Babe, Y. Zi, A. Guha, C. J. Anderson, and M. Q. Feldman, "How beginning programmers and code llms (mis)read each other," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI 24. ACM, May 2024, pp. 1–26. [Online]. Available: http://dx.doi.org/10.1145/3613904.3642706

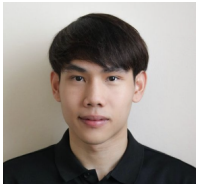
[8] W. Li, A. Marino, H. Yang, N. Meng, L. Li, and H. Cai, "How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3631967

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

[10] Council of Europe, "Common european framework of reference for languages: Level descriptions," 2020, accessed: 2025-04-10. [Online]. Available: https://www.coe.int/en/web/common-european-framework-reference-languages/level-descriptions

[11] G. Robles, R. G. Kula, C. Ragkhitwetsagul, T. Sakulniwat, K. Matsumoto, and J. M. Gonzalez-Barahona, "pycefr: Python competency level through code analysis," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC 22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 173–177. [Online]. Available: https://doi.org/10.1145/3524610.3527878

[12] N. Korkmaz, "10.000 english words cerf labelled," https://www.kaggle.com/datasets/nezahatkk/10-000-english-words-cerf-labelled/data, 2024, accessed: 2025-04-09.



**Ruksit Rojpaisarnkit** is a PhD student in the Software Engineering Laboratory at the Graduate School of Science and Technology, Nara Institute of Science and Technology (NAIST). His main research interests were related to the human aspect of software engineering, code proficiency, and data mining. Find him at https://www.linkedin.com/in/ruksit-rojpaisarnkit-76b72417a/.

**Youmei Fan** is an Assistant Professor in the Software Engineering Laboratory at the Graduate School of Science and Technology, Nara Institute of Science and Technology (NAIST). She completed a doctoral course in Software Engineering Laboratory at the Graduate School of Science and Technology, Nara Institute of Science and Technology (NAIST), Japan. During her doctoral degree, her main research interests were related to the human aspect of software engineering, open-source software sustainability, and data mining. Find her at https://www.linkedin.com/in/youmei-fan-513a161a6/.

**Kenichi Matsumoto** is a Professor in the Software Engineering Laboratory at the Graduate School of Science and Technology, Nara Institute of Science and Technology (NAIST) Contact him at matumoto@is.naist.jp.

**Raula Gaikovina Kula** is a Professor at The University of Osaka. He received his Ph.D. from Nara Institute of Science and Technology (NAIST) in 2013, later joining as a Specially-Apppointed Assistant Professor from (2013-2016) at Osaka University. He then continued as a Specially-Appointed Assistant Professor from (2017), later becoming an Assistant Professor (2017-2023), and a Associate Professor (2023-2024) at NAIST.