A Study of the Impact of AI-assisted Programming on the Work Efficiency of Software Engineering Developers

Wenxuan Zhang
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
1009230388

Renchao Wu
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
1009750672

Ruijie Shi
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
1009208787

***Abstract*—Artificial intelligence (AI) has become a widely utilized tool among real-world practitioners. Many popular AI platforms, such as ChatGPT, have been implemented to provide functionality for generating programming code. To comprehend the capabilities of these AI models in completing common programming tasks and their impact on the efficiency of software engineers, this paper proposes an empirical study on the efficiency of AI in completing common programming tasks. Our research results indicate that certain AI models exhibit excellent performance in code translation tasks, suggesting their potential to assist software engineers in enhancing their working efficiency in such tasks. Moreover, providing more specific prompts does not necessarily lead to higher accuracy in AI responses. Conversely, concise and accurate prompts are more effective in guiding AI towards correct discoveries and providing accurate answers. This finding can guide software engineers in designing prompt schemes and enhance their efficiency. The size of AI model parameters does not significantly affect their accuracy in completing code generation tasks. Selecting the appropriate model for code generation tasks may have a more significant impact on the efficiency of software engineers. In conclusion, our findings contribute to understanding the impact of AI models on the efficiency of software engineers when performing programming-related tasks, and they help guide software engineers in effectively utilizing AI models to improve work efficiency.**

***Index Terms*—Artificial intelligence, Software engineer, LLMs**

1. Introduction

The development of large code language models (code LLMs) such as Codex, Copilot and CodeBERT [6,7,8], along with significant advancements in machine learning models in the area of code generation, enable professional developers to synthesize new code snippets from existing code snippets or even natural language instructions [19]. The emergence of these new technologies is expected to significantly improve the efficiency of software development through the automation of the coding process [10]. However, despite the considerable potential of these technologies, their feasibility in practical application, the level of acceptance by developers (especially in terms of their practical ability to improve work efficiency), and the challenges and limitations they face are still important issues worth exploring in depth. The developer community shows varying levels of acceptance of artificial intelligence code generation technology [9]. On the one hand, developers have high expectations for the potential of this technology to improve programming efficiency and accelerate the code writing process. On the other hand, [10] researchers also expressed concerns about the accuracy, quality and potential impact on the code maintainability of the code. This study aims to investigate the capabilities of artificial intelligence in programming-related tasks and how they influence software programming efficiency, thus gaining insights into the impact of AI on the efficiency of software engineers in this domain. This includes identifying common types of programming tasks, analyzing the efficiency of AI in such tasks, and providing guidance for improving developers' work efficiency. Through this study, we hope to evaluate whether AI code generation technology can significantly improve the efficiency of software engineering, thereby providing insights for the future development of the software development field. The study is driven by three research questions (RQs):

RQ1: How does AI perform in cross-language translation programming? It is well known that there are many types of programming languages being used in real-world development environments today. Moreover, developers usually need to work in multiple programming language environments when facing actual work situations [15]. If a developer encounters the same programming problem in different programming languages, they would need to manually translate the available code from one programming language into a version usable in another programming language. This undoubtedly imposes a significant amount of repetitive labor on developers and diminishes their work efficiency. Therefore, we aim to explore whether AI is capable of shouldering this process for developers, thereby enhancing the efficiency of real-world developers.

RQ2: Can AI programming assistance applications be made more efficient when developers use more specific prompts? Typically, when we give clearer and more specific instructions to artificial intelligence such as ChatGPT, artificial intelligence may understand our purpose more clearly and give accurate answers. However, from the perspective of artificial intelligence and NLP, more complex descriptions often bring more vectors and variables that need to be fitted. Therefore, overly complex prompts may mislead artificial intelligence and make it lose focus, thereby providing developers with Unsuitable solution to the problem. What we want to focus on under this RQ is whether there is a prompt complexity or length interval that enables the AI code generation function to be efficiently utilized; or perhaps it is very straightforward, i.e., the more detailed the prompt the more likely that the AI's code-generating abilities will be fully utilized. Therefore, if software engineers can be provided with a set of accurate prompt design solutions, it will undoubtedly improve their work efficiency.

RQ3: Are parameter sizes a determining factor in LLM's code generation capabilities? From intuition and common sense related to NLP, larger parameter sizes tend to give LLMs a huge boost in capability such as the leap from GPT3 to GPT4. However, there is a difference between programming in code and writing in a human language, as programming tends to focus more on the application of logic and algorithms. If LLMs cannot understand the logic behind programming but simply splice the code according to a massive database, they may not be able to accurately complete the programming work even if they have a huge number of parameters, thus affecting their work efficiency. What we want to examine is whether a larger parameter size determines the ability of LLM code generation, which in turn determines its efficiency in real work. Therefore, if software engineers can choose the right model for their task it is equivalent to getting the right tool, which will increase their potential productivity.

2. Related Work

Software engineers often face a large number of repetitive tasks in their daily work, so being able to identify these repetitive tasks and automate them through AI technology is crucial to work efficiency. Therefore, researchers have developed corresponding technologies for different repetitive tasks, which allows artificial intelligence to complete the work in these fields, thereby freeing developers from the quagmire of repetitive tasks [4]. Tao et al. [1] invented a comparative cross-language code clone detection technology C4, which can effectively detect cross-language clones. The system utilizes a set of pre-trained models from the AI field. First, the authors collect relative dataset to construct clone and non-clone pairs, where they are considered cross-language clone pairs if there are different programming language solutions to a problem. Next, the authors use a fine-tuned CodeBERT [6] pre-trained model to obtain code representation, with the aim of obtaining better vector embeddings for code cloning downstream tasks. Finally, for the output results of the CodeBERT model, the author uses a combination of N-pair loss and soft nearest neighbor loss as the loss function. The parameters of CodeBERT were learned throughout the training process to minimize the contrastive loss and ultimately achieved an accuracy of 94% for checking cross-language clones.

In order to implement AI that can complete the task of code generation, the authors in [2] developed an approach based on large language models. They proposed a system called SantaCoder with 1.1 billion parameters that can accurately complete code generation tasks in multiple programming languages. First, their training raw data comes from a small subset of The Stack, which will be used as the training data set after preprocessing and multiple filtering. SantaCoder has two major components: tokenizer and decoder. They used Hugging Face Tokenizer as a tokenizer and used the byte-based Byte-Pair Encoding (BPE) algorithm to train on raw bytes. In addition, the training data used a digit splitter and GPT-2 pre-segmentation before being converted to bytes. The regular expression is pre-segmented. The decoder during the training process consists of a transformer with 1.1B parameters, which has FIM and MQA, and is trained with float16. SantaCoder offers significant performance improvements compared to previous technologies.

Niu et al. [3] also used a large language model to implement code generation, but the authors in [3] paid more attention to the pre-training of the decoder. Therefore, they proposed a sequence-to-sequence pre-training model for source code called SPT-Code. The architecture of SPT-Code is a multi-layer Transformer, and its parameter settings also follow the settings of CodeBERT. Its main features are two places: model input and three code-specific seq2seq pre-training tasks. For model input, the author extends the input representation of the pre-trained model, which consists of code tokens, linearized AST and natural language. Three pre-training tasks are the top priority of SPT-Code. The first one is Code-AST prediction, which allows the model to learn structural information from the X-SBT representation in the input. The second pre-training task is MASS, which allows the model to be pre-trained in a sequence-to-sequence manner. The last pre-training task is method name generation, which can help the model learn information such as the intent and function of the code, thereby strengthening the model's understanding of natural language. With these special configurations, SPT-Code is able to achieve excellent performance in five code-related downstream tasks.

Liu et al. [5] propose an evaluation framework that can automatically evaluate the code generation functions of several existing LLMs such as SantaCoder mentioned above. However, their research focused on the strict theoretical performance of LLM rather than an evaluation of the work efficiency of real developers using LLM to generate code.

Through understanding these cutting-edge technologies, we can find that more and more researchers are casting their sights on the field of artificial intelligence code generation

and developing corresponding technologies. And some researchers have begun to evaluate these artificial intelligence applications, but there is still a lack of sufficient investigation into real-world developers actual efficiency improvement by using these tools. Therefore, it is very worthy of attention to study whether the assistance of these technologies can make developer programming more efficient in actual work scenarios.

3. Methodology

To address the research questions outlined in the first section, we have currently collected three open-source datasets, namely DevGPT, llm-agents and CodeTrans [11,12,17]. DevGPT is a dataset related to ChatGPT activities sourced from GitHub, comprising 6 categories of snapshots and a total of 9948 usable relevant data entries. These data primarily consist of the processes and outcomes developers employ when using ChatGPT to assist in programming. The second dataset, llm-agents, is sourced from the Hugging Face open database, from which we selected 463 LLM code generation-related data out of 3824 entries. These records document how various types of LLMs handle issues related to code generation and their resulting feedback. CodeTrans [17] also comes from an open source Github repository, which contains the performance of 5 common AI models when performing Java and C# programming language translation tasks. And the performance of each AI model in the data set has been evaluated by three baseline methods. The following describes our data collection methodology, with the aim of gathering feedback from real developers using artificial intelligence code generation functionality, along with detailed specifics of their usage such as prompt frequency and length. Subsequently, we applied analysis method to both datasets to complete quantitative analysis in order to explore the efficiency of code generation functionality in real-world tasks. Figure 1 outlines our approach.

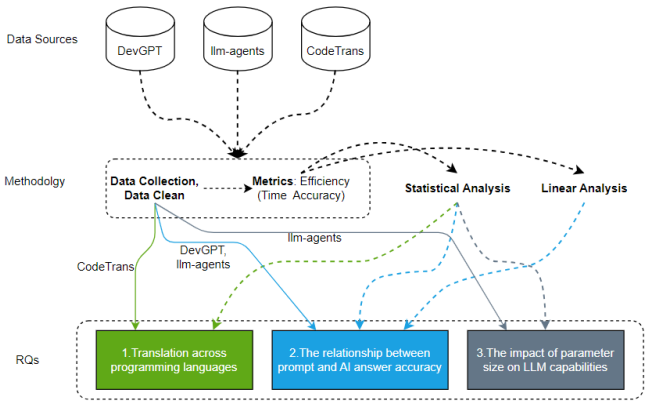

Figure 1. Overview of the methodology

3.1. Data Collection

For the two datasets, we followed different collection procedures. For DevGPT, we initially downloaded the complete dataset consisting of records from 9 time periods from its GitHub page. Subsequently, we underwent a data cleaning process, discarding entries unrelated to the main theme and excluding those containing error information such as "404 not found" network interruption information, which could not be utilized in subsequent analysis steps, such as entries solely documenting interruptions in ChatGPT network connections. Regarding llm-agents, we selected one dataset from the 29 datasets gathered in prior research [5]. This selection was made because among these datasets, llm-agents focused more on the outcomes of LLMs when confronted with various tasks and their success or failure. Furthermore, we performed additional filtering on the llm-agents dataset, excluding other task types such as arithmetic, and retained only data records pertaining to LLMs executing code generation tasks. For CodeTrans, we selected one of the 14 data sets tested and collected in previous research [16], because CodeTrans mainly collects data samples of mutual translation tasks between different programming languages. At the same time, we conducted a preliminary check to confirm that the data set can be run and analyzed normally.

3.2. Efficiency Evalution Metrics And Analysis Methods

In order to quantitatively analyze our topic of "efficiency", we first need to define metrics to determine what constitutes efficiency and quantify it.

Initially, we define the criteria for assessing the efficiency of AI-assisted code programming functionalities in two dimensions: time and accuracy. Time refers to how quickly AI can provide developers with viable code solutions. In this dimension, we will disregard network speed and the speed at which AI responds, as both are external to the AI model itself. Network speed is influenced by the developer's network environment, while the speed at which AI returns answers is determined by its internal settings, such as how quickly ChatGPT displays characters on the screen. Therefore, in the time dimension, our main concern is how quickly AI can provide developers with the correct target solutions within a few prompts. Accuracy, on the other hand, refers to whether the answers provided by AI to developers are as expected, such as whether they are in the correct language or follow the desired code format specified by the developer.

Next, for the quantitative analysis steps, we have employed a linear analysis method to examine potential linear relationships among various variables. Additionally, we will also focus on measuring the multiple variables statistically.

4. Results

4.1. Statistical evaluation of AI on Code2Code translation programming tasks (RQ1)

To address RQ1, we will analyze the performance of AI in cross-language programming from the perspective of accuracy using the CodeTrans dataset. The CodeTrans dataset contains performance results of five AI methods - Naive copy, PBSMT, Transformer, Roberta, and CodeBERT - on code translation tasks across languages. The performance evaluation of these five methods is based on three baseline metrics: BLEU scores, accuracy, and CodeBLEU scores [18], which measure the similarity of machine-translated text results to high-quality developer translation samples. In this section, we can assess the performance of AI in cross-language programming from the accuracy dimension using these three baseline statistical indicators.

Tables 1 and 2 record the performance of various AI methods in performing translation tasks between Java and C# languages. From the two tables, we can clearly observe that the Roberta and CodeBERT methods among the five tested methods have already demonstrated excellent performance in this task. This indicates that under these two AI methods, the translation ability of AI is approaching the level of real developers. Specifically, CodeBERT maintains a CodeBLEU score of 85.1 when translating from Java to C#, while Roberta achieves a CodeBLEU score of 80.18 when translating from C# to Java. Meanwhile, we also observe that using the simple Naive copy method, AI completely fails to perform code translation work, as this method performs poorly in all metrics.

Method	BLEU Score	Accuracy	CodeBLEU Socre
Naive copy	18.54	0	-
PBSMT	43.53	12.5	42.71
Transformer	55.84	33.0	63.74
Roborta	77.46	56.1	83.07
CodeBERT	79.92	59.0	85.10

Table 1: Performance of Java to C# translation.

Method	BLEU Score	Accuracy	CodeBLEU Socre
Naive copy	18.69	0	-
PBSMT	40.06	16.1	43.48
Transformer	50.47	37.9	61.59
Roborta	71.99	57.9	80.18
CodeBERT	72.14	58.0	79.41

Table 2: Performance of C# to Java translation.

Based on the statistical results of the three indicators mentioned above, we can conclude that some AI methods do have the ability to accomplish code translation tasks, and under certain AI methods, their performance approaches that of real developers. This outstanding performance may be attributed to the fact that code translation tasks require AI to master the syntax aspects of programming languages rather than fully understanding the logic within the code. Additionally, we notice slight differences in performance between Java to C# and C# to Java translation tasks, but the discrepancies are not significant, possibly due to slight imbalances in the training data samples. According to the aforementioned results, the tested AI methods are indeed capable of relieving real developers of repetitive work in crosslanguage code translation to some extent. This may assist real developers in focusing more on other critical programming tasks such as algorithm design, thereby improving their potential work efficiency.

4.2. Linear analysis of Prompt specificity and artificial intelligence accuracy (RQ2)

To address RQ2, we will conduct an analysis from two dimensions: time and accuracy. Firstly, from the perspective of time, we will utilize data from DevGPT to provide insights. In order to gauge the efficiency of AI responses, we have established the following criteria: if developers need to modify prompts multiple times and resubmit them to the AI to obtain the correct answer, we consider this process to be inefficient. Conversely, if the correct answer can be obtained with just one prompt, we consider it to be an efficient process. To measure the specificity of prompts, we use the length of prompt tokens as an indicator. Longer and more detailed prompts typically require developers to provide lengthier descriptions, thus, in this experiment, prompts with longer token lengths are considered more specific, while shorter ones are considered more concise.

Figure 2 depicts a line graph illustrating the linear relationship between token length and the number of prompt modifications. From the graph, it is evident that even as the token length of prompts increases, there is no significant decrease in the number of modifications and resubmissions required. On the contrary, ChatGPT needs to get the prompt modified more times in order to get the correct solution that the developer expects.

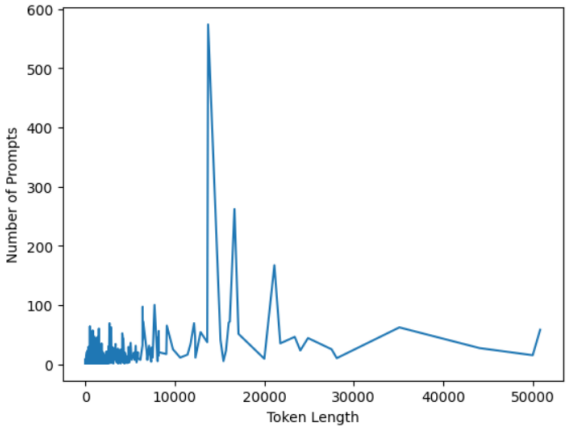

Figure 2. Number of prompt vs Token length of prompt

Similarly, the statistical data from the DevGPT dataset also corroborates this phenomenon. Figures 3 and 4 respec-

tively represent this relationship in the dataset's first and second quartiles. We observe that the majority of ChatGPT instances require very short prompt token lengths to generate code correctly on the first attempt. Additionally, at least half of the instances in the dataset can generate correct responses within two attempts, with the token length of prompts used still remaining at a relatively short level.

NumberOfPrompts	1.0
TokensOfPrompts	55.0

Figure 3. First quartiles of dataset

NumberOfPrompts	2.0
TokensOfPrompts	266.0

Figure 4. Second quartiles of dataset

This indicates that ChatGPT often requires concise and precise prompts to generate solutions correctly during code generation tasks. More specific prompts may confuse ChatGPT about the focus of the problem, thus leading to the generation of incorrect answers.

Meanwhile, from the perspective of accuracy, data from llm-agents also supports this viewpoint. When examining accuracy, we focus on the length of prompts used by developers in LLM's correct or incorrect responses. The scatter plot in Figure 4 illustrates the performance of the llm-agents dataset in this dimension. Points on the y-axis at 1 represent correct answers, while points at -1 represent incorrect answers.

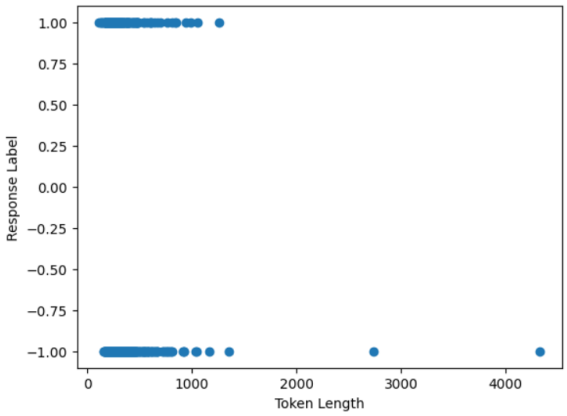

Figure 5. Llm-agents response label

From Figure 5, it is evident that within a certain range of prompt token lengths, increasing token length does not significantly improve the accuracy of code generation by LLM. Therefore, we can conclude that providing more detailed prompts does not necessarily help developers achieve higher accuracy or greater efficiency in their work.

Combining the findings from both datasets and dimensions, we can infer that the level of detail in prompts indeed impacts the efficiency of AI in completing code generation tasks, but it does so negatively. Developers using AI for programming should strive to describe their problems in concise and precise language to enable AI to perform code generation tasks more efficiently. Instead of hoping to provide AI with every small detail of the problem, which may mislead it and ultimately decrease work efficiency.

4.3. Statistical analysis of parameter size on LLM code generation capability (RQ3)

To address RQ3, we will analyze the llm-agents dataset from the perspective of accuracy. The llm-agents dataset comprises various types of LLMs' performances in executing code generation tasks. The programming tasks executed in the llm-agents dataset are sourced from HumanEval and MBPP baselines [13,14], with the difficulty level of the programming tasks suitable for entry-level programmers to solve. Therefore, through statistical analysis of this dataset, we can gain a clear understanding of the capabilities and efficiency of various LLMs in handling basic problems in developers' daily work. In this section's analysis, we can directly obtain the performance of different types of LLMs from the response_label parameter in the dataset.

Figures 6 and 7 respectively record the number of correct responses and the accuracy rate of various LLMs in executing code generation tasks. Combining the two figures, we can clearly observe that, contrary to our intuition, LLMs with a large number of parameters do not exhibit a significant performance advantage when executing programming tasks. From the figures, we can see that the LLaMa-2-13b model performs best on this task, despite having only 13 billion parameters. In contrast, the GPT-4 model, which has the largest number of parameters among the listed model types (1.76 trillion), does not demonstrate outstanding performance. In fact, it even has a lower task execution accuracy rate than the LLaMa-2-13b model by 10%. On the other hand, the worst-performing model, Vicuna-33b, has 33 billion parameters, with an accuracy rate of only 15%.

Combining the aforementioned statistical analysis in terms of accuracy, we can deduce that the influence of parameter size on LLM code generation abilities is not as significant as we might have imagined. Therefore, there may be other underlying factors that determine LLM programming capabilities. These factors could stem from the composition of the data used to train LLMs, with more emphasis on human language text samples and insufficient programming samples. Alternatively, it could originate from researchers needing to employ special techniques to enable LLMs to understand programming logic and algorithms. In summary, this poses a potential issue for future research to enhance LLM programming capabilities.

Meanwhile, it's worth besides noting that the code generation accuracy of all LLMs surveyed in this investigation remains at a relatively low level. Even the best-performing tested model achieves only 40% accuracy. This indicates that

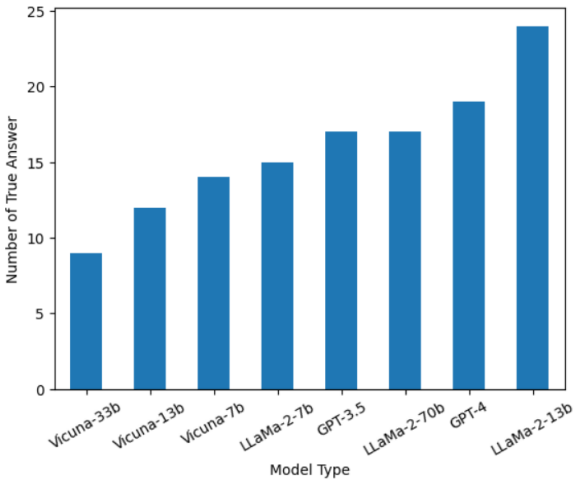

Figure 6. Number of true answer

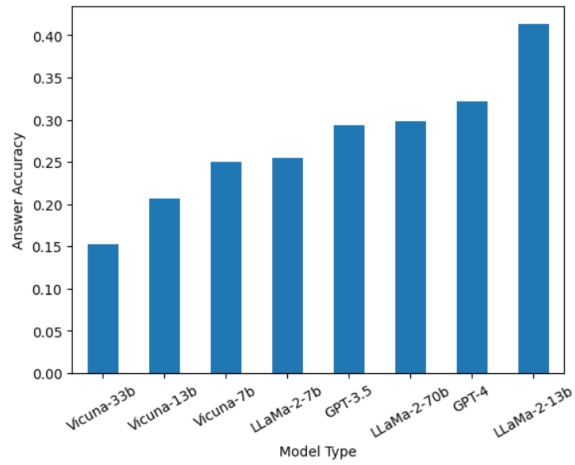

Figure 7. Answer accuracy

LLMs still offer limited assistance in improving the working efficiency of today's real-world developers. Therefore, further research is needed to make LLMs more practically beneficial for developers' daily programming tasks.

5. Discussion

Our research objective is to explore the potential of AI code generator in improving developers programming efficiency. Our analysis encompassed the performance of AI in code-to-code translation, the efficiency implications of prompt specificity, and the influence of LLM parameter sizes on code generation capabilities.

The power and impact of AI cross-programming language translation. Translating code from one programming language into a usable form in another programming language is a common job for developers. This is a study that statistically evaluates the ability of these common AI models to do this type of work, taking into account multiple baseline metrics. A clear understanding of AI's ability to translate code is important to developers' work efficiency. Our findings can help developers better understand which AI-assisted tools can be used to simplify the development process and improve work efficiency.

The relationship between prompt and AI code generation efficiency. Our research provides an in-depth analysis of the relationship between prompt detail level and AI code generation efficiency. These relationships may determine developers' strategies for designing prompts when communicating with AI. Contrary to the our expectation that more detailed prompts would bring better AI performance, our findings indicate that the efficiency of AI in code generation tasks may not linearly related to the complexity of prompts. In fact, overly cumbersome prompts may surprisingly decrease AI efficiency, an accurate and concise prompt can lead to better performance. This discovery can help developers choose strategies for designing prompts to efficiently use AI for code generation.

The impact of parameter size on LLM code generation capabilities. We counted and evaluated the performance results of several common LLMs when performing code generation tasks. These results are counter-intuitive that the parameter size of the model is not a decisive factor in the LLM capability for this kind of work. There are other factors potentially affecting LLM's ability in this task that deserve our future exploration. Besides, the accuracy of all LLMs is on a relatively low level. This suggests that the current code generation capabilities of LLM are not enough to support it in generally sharing the programming work of developers, and the improvement in work efficiency achieved by developers using LLM tools is still limited.

6. Threats to Validity

Our study is exposed to several factors that could limit the generalizability and validity of the findings. Firstly, the dataset we used may not fully represent the diverse range of programming tasks. We used DevGPT, llm-agent and CodeTrans as our primary dataset, each of them only represents a small subset of different AI code generator tasks, so our finding may be constrained by the specific datasets and AI models. Besides, we have discarded some data which are hard to quantify, and only used a part of data in three dataset. Different artificial intelligence models may have different characteristics and special areas of expertise., which could affect their performance in code generation tasks. Therefore, the evaluation of the efficiency of each AI model in this study may be biased which may not represent the use of all exist AI models in these tasks.

Secondly, real world software development scenarios may differ from our assumption. Our study primarily focused on isolated programming tasks such as a single code generation inquiry to ChatGPT, on the contrary, part software development projects may involve the exposure of more information and the linkage of more code modules. Therefore, Our evaluation may not be fully applicable to represent the performance of AI in this situation.

Furthermore, the metrics we use to define and measure the efficiency of AI code generators may not be granular enough to be applicable to all scenarios. There may be room for broader expansion of our considerations in terms of time and accuracy metrics. This limitation may cause some variation in our estimates of AI model performance. A more generalized study might need more objective and exhaustive metrics. Unfortunately based on available datasets and project time constraints, we cannot study it in fully detail here.

Lastly, our conclusion may have certain limitations on statistical analysis. The linear and statistical analysis methods employed to interpret the data may not capture the complex, non-linear relationships between variables.

7. Limitations and Future Work

In the research conducted for this project, our findings primarily stem from quantitative analysis of existing data. However, insights regarding the impact of using AI on the efficiency of software engineers, derived from feedback provided by software engineers themselves, may offer a more realistic perspective on actual work environments. Due to constraints imposed by the duration of this project and the challenges associated with obtaining Research Ethics Board (REB) approval, we were unable to gather genuine opinions and perspectives from software engineers. Therefore, in future research endeavors, obtaining real-time data through surveys and interviews with software engineers would enhance the applicability of our research findings to real-world work scenarios. Additionally, expanding the available datasets in future research will contribute to enhancing the reliability of our quantitative analysis, thereby improving the validity of our findings. Finally, exploring more detailed qualitative analysis evaluation metrics in future work is also meaningful. This would allow our qualitative research results to encompass a broader spectrum of domains and align with a variety of work scenarios.

8. Conclusion

This paper presents an empirical study on the impact of AI-assisted programming tasks on the efficiency of software engineers, revealing the capabilities and efficiency of AI models in completing code translation tasks. Our findings underscore the importance of correctly designing prompts for communication with AI models, which can enhance the accuracy of AI responses and improve the work efficiency of software engineers. Furthermore, the research results indicate that the size of model parameters does not significantly affect the accuracy of code generation tasks, thereby guiding software engineers to adopt more suitable AI models for programming tasks to enhance their work efficiency. In summary, our work contributes to guiding software engineers in selecting and utilizing AI, thereby improving their potential work efficiency by identifying the impact of AI on the efficiency of software engineers.

References

[1] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive Cross-Language Code Clone Detection," ACM Reference Format, vol. 12, 2022, doi: https://doi.org/10.1145/3524610.3527911.

[2] L. B. Allal et al., "SantaCoder: don't reach for the stars!," arXiv.org, Feb. 24, 2023. https://arxiv.org/abs/2301.03988

[3] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations," 2022, doi: https://doi.org/10.1145/3510003.3510096.

[4] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review," Entropy, vol. 25, no. 6, p. 888, Jun. 2023, doi: https://doi.org/10.3390/e25060888.

[5] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," arXiv.org, Jun. 12, 2023. https://arxiv.org/abs/2305.01210

[6] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Empirical Methods in Natural Language Processing, Feb. 2020, doi: https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[7] W. Zaremba and G. Brockman, "OpenAI Codex," openai.com, Aug. 10, 2021. https://openai.com/blog/openai-codex

[8] GitHub, "GitHub Copilot · Your AI pair programmer," GitHub, 2023. https://github.com/features/copilot

[9] S. Kelly, S.-A. Kaye, and O. Oviedo-Trespalacios, "What Factors Contribute to Acceptance of Artificial Intelligence? A Systematic Review," Telematics and Informatics, vol. 77, no. 77, p. 101925, Dec. 2022, doi: https://doi.org/10.1016/j.tele.2022.101925.

[10] D. Russo, "Navigating the Complexity of Generative AI Adoption in Software Engineering," ACM Trans. Softw. Eng. Methodol, vol. 37, no. 111, 2024, doi: https://doi.org/10.1145/1122445.1122456.

[11] "Hugging Face – The AI community building the future.," huggingface.co, Jan. 04, 2024. https://huggingface.co/datasets?other=code-generation (accessed Mar. 05, 2024).

[12] "DevGPT: Studying Developer-ChatGPT Conversations," GitHub, Dec. 04, 2023. https://github.com/NAIST-SE/DevGPT/tree/main

[13] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.

[14] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv preprint arXiv:2303.17568, 2023.

[15] "What Is a Full Stack Developer & What Do They Do?," UofT SCS Boot Camps, Nov. 11, 2020. https://bootcamp.learn.utoronto.ca/blog/what-is-a-full-stack-developer/

[16] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664, 2021.

[17] "CodeXGLUE/Code-Code/code-to-code-trans at main · microsoft/CodeXGLUE," GitHub. https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans

[18] S. Ren et al., "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," arXiv (Cornell University), Sep. 2020.

[19] F. F. Xu, B. Vasilescu, and G. Neubig, "In-IDE Code Generation from Natural Language: Promise and Challenges," ACM Transactions on Software Engineering and Methodology, vol. 31, no. 2, pp. 1–47, Apr. 2022, doi: https://doi.org/10.1145/3487569.