Code Comprehension: Review and Large Language Models Exploration

2024 IEEE 4th International Conference on Software Engineering and Artificial Intelligence (SEAI) | 979-8-3503-7434-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/SEAI62072.2024.10674263

Jielun Cui
Computer Science
University of Cincinnati
Cincinnati, USA
cuiju@mail.uc.edu

Yutong Zhao
Computer Science and Cybersecurity
University of Central Missouri
Warrensburg, USA
yutongzhao@ucmo.edu

Chong Yu
Computer Science
University of Cincinnati
Cincinnati, USA
yuc5@ucmail.uc.edu

Jiaqi Huang
Computer Science and Cybersecurity
University of Central Missouri
Warrensburg, USA
jhuang@ucmo.edu

Yuanyuan Wu
Computer Science
University of Cincinnati
Cincinnati, USA
wu3yy@mail.uc.edu

Yu Zhao
Computer Science
University of Cincinnati
Cincinnati, USA
zhao3y3@ucmail.uc.edu

***Abstract*—This paper presents a comprehensive survey of code comprehension, a crucial aspect of Software Engineering that encompasses the process by which developers understand code. Our study categorizes code comprehension research into four areas: code comment generation, the correlation of EEG signals with code comprehension, experimental studies on code comprehension, and code visualization, detailing methodologies within each. Further, we investigate the role of Large Language Models (LLMs) in enhancing code comprehension tasks, highlighting the potential future research opportunities. This research aims to provide software engineers and researchers with a comprehensive understanding of the current technologies for code comprehension and to point out possible directions for future research.**

Keywords—Code comprehension, LLMs, comment generation

I. INTRODUCTION

Code Comprehension is a fundamental concept in the domains of Software Engineering. It describes the process through which developers interpret and understand the meaning of code. This includes not only the direct grasp of the code's semantics but also an understanding of the underlying design goals, architecture, and significant potential impacts. Such comprehension is crucial for various aspects of software development, including the need for developers to reuse legacy code written by others and software maintenance requiring updates/fixes to existing code. A study [1] found that about half of software engineers leave their jobs every two years. This frequent job switching makes it even more important to understand other developer's code quickly and efficiently. Moreover, software maintenance, a critical stage of the Software Development Life Cycle (SDLC), can account for up to a huge percentage of a software's total cost of ownership [2]. With an estimated 70% of a developer's time spent understanding code [3], there's a strong incentive for software companies and developers to develop strategies for more efficient code comprehension solutions to reduce costs.

This paper aims to offer a thorough review of code comprehension, detailing the distribution and trends of research across various categories. It is structured around four key research fields related to code comprehension: code comment generation [4]-[7], the correlation between electroencephalogram (EEG) signals and code comprehension [8]-[11], experimental studies on code comprehension [12]-[14], and code visualization [15]-[18]. For each category, we will present a selection of representative approaches, accompanied by an in-depth discussion of their methodologies.

In addition to analyzing existing literature, our study leverages the innovative capabilities of Large Language Models (LLMs) to assist in code comprehension. The emergence of LLMs — pre-trained models enriched with vast amounts of real-world data, exemplified by ChatGPT [19]—has notably advanced progress across various software engineering tasks in recent years. For instance, Liu et al. have employed LLMs to generate test inputs for Android applications [20]. In our work, we design specific prompts to investigate the current capacity of LLMs in code comprehension tasks, aiming to uncover future research opportunities. Some existing works have previously explored methods for generating source code comments, such as the study by Xiaotao et al. [21]. However, this particular study was published in 2019, a period during which Large Language Models (LLMs) had not yet demonstrated their current level of efficiency, resulting in their exclusion from the analysis. Another work Codex [22] is a GPT-3 based approach for automating the generation of code comments. Nevertheless, this approach was limited to generating comments at the function

level. In contrast, our current work leverages the latest version of GPT, namely GPT-4, to design a variety of prompts that facilitate comment generation at two granularities: within individual lines of code and at the function level. Furthermore, we employ Meteor [23] as the metric for assessing comment generation performance, replacing BLEU [24]. This decision is based on Meteor's ability to recognize synonymous words that differ in appearance, offering a more nuanced evaluation of semantic accuracy. Moreover, we extend the use of GPT-4 beyond comment generation, employing it to investigate additional code comprehension functionalities, such as code visualization.

In summary, this paper makes the following contributions:

1. A comprehensive survey of four research fields related to code comprehension approaches.

2. An exploration into the use of LLMs for code comprehension tasks.

3. The availability of the LLM-based code comprehension tasks discussed in this paper, alongside all experimental data, for public access [25].

II. Related Work

In addition to the related works discussed in Sections I, other studies also explore the use of LLMs in software engineering. For example, a recent study [29] investigate the use of LLMs in code generation.

III. Study of Code Comprehension

In this study, we will survey the tools that have been employed in the domain of code comprehension in the past. In terms of application, our emphasis will be on four research categories of comprehending code: code comments, EEG, experimental comprehension, and visualization.

A. Code Comment Generation

A code comment describes the logic and function behind the source code, translating complex code into understandable natural language. Comprehensive comments can improve the readability of the project, thereby positively impacting software maintenance, facilitating software reuse, and benefiting various other facets of software engineering and related fields.

To generate code comments efficiently and accurately, Wong et al. [4] applied Question and Answer (Q&A) sites for automatic comment generation (AutoComment). By leveraging Q&A sites, they can post a couple of test questions and receive the code-description mappings that contain both code segments and descriptions. With these code-description mappings, comments can be automatically generated for open-source projects by matching the code segments. To validate the proposed approach, this work analyzed Java and Android tagged Q&A posts and extracted 132,767 code-description mappings. Utilizing these mappings, 102 comments were generated for 23 Java and Android projects. This study provides valuable insights into the area of code comment generation, shedding light on effective strategies, challenges, and potential advancements in enhancing code readability and understanding. However, a limitation of this work is its reliance on existing templates, especially since templates for specialized functions may not always be available.

To eliminate reliance on templates, Liang et al. [5] developed a comment generation framework consisting of Code RNN and Code-GRU. In this framework, Code RNN is an arbitrary tree, where each program's parse tree is encoded into a neural network and each syntactic node is represented by a vector, thereby describing the structural information of source code. Code-GRU, a variant of Recurrent Neural Networks (RNNs), incorporates an additional choice gate, enabling it to directly handle the representation vectors of code blocks. When vector representations generated by Code RNN are fed into a Code-GRU, the entire framework effectively produces text descriptions of the code. This study utilized 10 open-source Java code repositories sourced from GitHub for performance validation. Compared with learning-based approaches, it achieved superior accuracy in code comments. This study offers a compatibility framework applicable to a range of programming languages, provided we have access to the parse tree of the input program.

Loyola et al. [6] leveraged both code commits and intra-code documentation to produce informative and concise descriptions. They assumed that these two types of docstrings are interdependent, as any alterations to the code are expected to be reflected in its functionality. Based on this hypothesis, they designed an architecture that merges change descriptions with source code documentation, utilizing their relationship to guide the generation of comment descriptions. For the purpose of performance validation, this study created a dataset containing change history and docstring data from various real-world open-source Python projects. The results show that considering signals from the content of the source code file contributes to improving the quality of comments. The finding suggests that further research considering the generation of descriptions from software artifacts from a more systemic perspective may be warranted.

B. EEG

The integration of EEG technology with machine learning and computational tools has opened new avenues in understanding the cognitive aspects of software development and code comprehension. This cutting-edge approach to neuroscientific research is increasingly being recognized for its potential to decode the complex neural activities underlying programming tasks, from code generation to debugging. By leveraging EEG data in conjunction with advanced analytical methods, researchers are embarking on a journey to uncover the cognitive processes that drive software engineering. This section is dedicated to exploring the innovative efforts, which combine EEG technology with diverse computational strategies, aiming to illuminate the neural mechanisms that support software engineering tasks and ultimately enhance our grasp of the cognitive dynamics involved in programming.

Lin et al. [10] set the foundation by exploring cognition during program comprehension through EEG activities and eye movements. They analyzed how 33 computer science undergraduates' cognitive processes, such as working memory and attention allocation, affect their performance in

programming tasks. The experiments read two programming tasks while their EEG activities and eye movements were recorded. The findings indicated that participants with high performance demonstrated superior working memory, as evidenced by increased theta power, more efficient allocation of attention resources (shown by reduced alpha power), and enhanced collaboration between working memory and semantic memory (reflected in elevated alpha power), during the understanding of complex programming constructs. This study not only highlights the cognitive roles in program comprehension but also offers insights for designing effective pedagogical strategies.

Building upon this foundational knowledge, Ishida and Uwano [9] investigated the synchronization of eye movements and EEG activities during program comprehension tasks. They measured brain waves and eye movements of programmers as they comprehended source code to analyze differences in time-series brain wave features between successful and unsuccessful comprehension. They discovered that participants who understood the code demonstrated significant increases in the α wave power spectrum and shifted their focus from specifications to source code more rapidly. This investigation into synchronized analysis suggests a method for real-time detection of programmers' comprehension.

Gonçales et al. [8] expanded the exploration by assessing the potential of EEG data, combined with machine learning classifiers, to distinguish developers' code comprehension levels. Utilizing a dataset from 35 developers undertaking code comprehension tasks, this study evaluates K-Nearest Neighbor (KNN), Neural Network (NN), Naïve Bayes (NB), Random Forest (RF), and Support Vector Machine (SVM) classifiers. Results highlight the KNN classifier's superior performance, achieving an 86% f-measure mean compared to the other methods with 80%. This study underscores the potential of using EEG data to classify code comprehension, suggesting a paradigm shift towards integrating psychophysiological data in software engineering to enhance task assignment and assess code quality, presenting a significant leap towards understanding the cognitive process of code comprehension through ML and EEG data.

In a subsequent study, Gonçales et al. [11] further refined the methodology by delving into the impact of filtering EEG signals on the classification of developers' code comprehension. The study enhances the precision of classification by employing both high and low pass filtering techniques designed with a Finite Impulse Response (FIR) filter using a Hamming window, aimed at removing noise and artifacts not pertinent to the cognitive processes involved in programming tasks. This meticulous process also encompassed the removal of abnormal signals and the application of Independent Component Analysis (ICA) through the fast ICA method, specifically to eliminate eye movement artifacts, thereby purifying the EEG data for analysis. The findings revealed a notable improvement in classification accuracy for the model trained on filtered EEG data, underscoring the efficacy of sophisticated EEG signal filtering in augmenting the precision of machine learning-based classification of developers' code comprehension.

C. Experimental Comprehension

To delve into the complexities of software readability and comprehension, Borsteler et al. [13] embarked on a study examining the influence of method chains and code comments through a detailed experimental analysis. Their investigation utilized code snippets that varied in the presence of method chains (with or without) and the nature of code comments (good, bad, or none). The participants, comprising first and second-year Computer Science students of varied coding backgrounds, were exposed to a series of code snippets in two rounds, tasked with evaluating the code and completing a cloze test. Analyzing data from 104 subjects, their findings revealed that while statement-level code comments impacted software readability, they did not significantly affect comprehension. Similarly, the presence or absence of method chains showed no substantial correlation with either readability or comprehension.

In a separate study, Swidan et al. [12] explored the "Reading Code Aloud" technique's effect on comprehension, engaging 49 primary school students in their experiment. The students were divided into a control group of 24 and an experimental group of 25, with both groups receiving three programming lessons lasting 1.5 hours each. Unlike the control group, the experimental group was instructed to read the code aloud collectively following the instructor. The evaluation, informed by Bloom's taxonomy through an 11-question assessment, sought to gauge the students' understanding and comprehension levels. The outcomes suggested that while "Reading Code Aloud" enhanced code memorization, it did not significantly improve comprehension. These results indicate the potential of the method in educational settings, though they prompt further investigation with a larger sample size to validate the experiment's reliability.

To explore the broader landscape of code comprehension research, Wyrich et al. [14] took a systematic mapping study, in which 95 experiments from 1979 to 2019 were reviewed. This comprehensive analysis aimed to understand how various studies approached the challenge of measuring code comprehension, focusing on experimental designs, participant demographics, and the nature of code snippets used. The study discovered a predominant use of within-subject designs and a shift towards using Java as the principal programming language in experiments. It also identified significant research themes, including the impact of semantic cues and developer characteristics on comprehension. The findings suggest a need for more standardized experiment designs and reporting to enhance the comparability and reliability of future code comprehension research. This mapping study provides a foundational resource for researchers by consolidating design characteristics and highlighting prevalent issues and opportunities for advancing the field of code comprehension.

D. Visualization

In the ever-evolving field of software development, the quest to enhance program comprehension through innovative visualization tools has led to a series of impactful studies, which illustrate a progressive journey toward understanding the multifaceted ways in which visual aids can improve our grasp of complex software systems.

Cornelissen et al. [15] initiated this exploration by conducting a controlled experiment to quantitatively evaluate

the effectiveness of trace visualization, particularly through their tool EXTRAVIS, in enhancing program comprehension. EXTRAVIS offers interactive visualization of execution traces, including a massive sequence view and a circular bundle view, enabling users to grasp the program's global structure and behavior efficiently. The study showed statistically significant benefits in employing trace visualization, with users experiencing a 22% reduction in time required and a 43% improvement in task accuracy. Building on this foundation.

Building upon this foundational insight into the benefits of visualization, Asenov et al. [16] expanded the scope by investigating the impact of enriched code visualizations on program comprehension through a user study involving 33 developers. This research compared traditional syntax highlighting with enhanced visual presentations, demonstrating that richer visualizations could significantly reduce the time needed to answer code-related questions without causing visual overload. Their findings recommend increasing the visual variety in code presentations to boost developers' efficiency and understanding.

Further advancing the conversation on the utility of specific visualization techniques, Umphress et al. [18] delved into the Control Structure Diagram (CSD) and the Complexity Profile Graph (CPG), demonstrating their roles in enhancing and measuring software code comprehensibility. The CSD, by overlaying graphical notations on source code, significantly improved comprehension among students, enabling more accurate and swift answers about the code. Conversely, the CPG, aiming to depict the relative complexity of code segments, showed its potential as a comprehensibility predictor by correlating complexity measures with the time required for correct understanding, despite no significant correlation with error rates.

Yin and Keller [17] tied these thematic threads together by introducing “Visualization in Contexts”, a strategy that leverages tools like the Context Viewer within the SPOOL environment to foster program comprehension across various contexts and abstraction levels. This comprehensive strategy supports the formation and integration of diverse mental models through visualization, showcasing the potential of tools like the Context Viewer to significantly improve the efficiency and accuracy in understanding complex software systems. Together, these studies paint a progressive picture of the evolving landscape of software visualization tools and strategies, from foundational insights into trace visualization's benefits to the comprehensive application.

IV. EVALUATION

To explore the potential of utilizing LLMs in enhancing code comprehension, we developed two primary strategies: comment generation and code visualization. Please note that our objective is not to evaluate the overall performance of LLMs across tasks within a large dataset. Instead, we are conducting a thorough exploration to assess the potential performance of LLMs in code comprehension.

To assess the capability of LLMs in generating code, we opted not to use the widely recognized Codexglue dataset [26]. This decision was driven by the following rationale: The Codexglue dataset consists of a vast amount of commented source code from real-world industry software, including numerous definitions and web API uses. However, our interest primarily lies in exploring the most challenging aspects of code comprehension, such as understanding code algorithms. Therefore, we randomly selected 9 algorithmic code samples from GeeksforGeeks [27], evenly distributed across difficulty levels with 3 easy, 3 medium, and 3 hard questions, to better align with our research focus. For our evaluation, we employed the Meteor [23], a widely recognized metric for assessing machine-generated text.

A. Comment Generation

For comment generation, we crafted two GPT prompts tailored to different levels of detail. The first prompt, "Please generate comments after each line of code," is designed for generating line-specific comments. The second, "Please generate a description for the whole function," aims to provide an overall description on entire functions.

In our dataset of algorithmic code, the BLEU scores approach zero due to the presence of synonymous words generated by GPT-4.0 offering the same meaning which affect the precision of comment generation evaluation. The evaluation results using the METEOR metric are as follows: METEOR scores for easy, medium, and hard code comment generation tasks are 0.24/0.2/0.18, respectively. Similarly, the scores for easy, medium, and hard function description generation tasks are 0.34/0.30/0.30. These findings suggest that comment generation and function description generation exhibit acceptable performance levels.

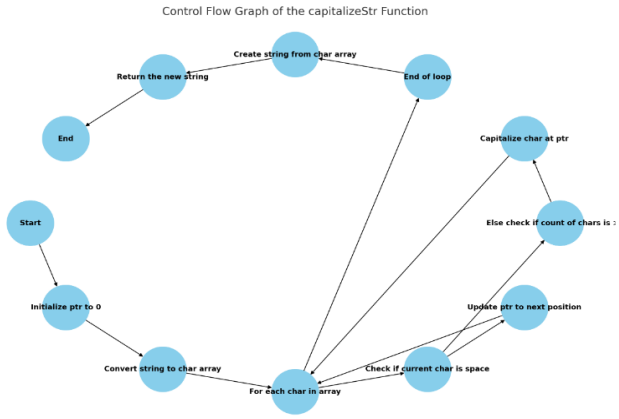

Fig. 1. GPT generated CFG

B. Code Visualization

In our study, we employ GPT 4.0 to generate Control Flow Graphs (CFGs) — a foundational graphical representation of code for code visualization purposes. After experimenting with various prompts, we find the most effective one to be: "Could you generate a control flow graph of the code in matplotlib format, ensuring that the nodes and edges do not overlap? On each node of the control flow graph, please use a short explanation sentence rather than raw code." An illustrative example is the code snippet titled "Capitalize the 1st character of all words having at least K characters" from GeeksforGeeks [28], which is used to generate a CFG as shown in Fig. 1.

In the resulting CFG, GPT 4.0 adeptly identifies each line of code and its structure, utilizing the code's functional essence to label the nodes within the CFG. This approach, facilitated by GPT 4.0, significantly enhances code comprehension by providing a clear, structured visual representation. Upon manually reviewing all the code snippets in our dataset, we noted that GPT-4.0 was able to accurately and clearly generate CFG for 7 out of 9 code snippets. One CFG generated from medium code is incorrect due to errors within the generated CFG. Furthermore, one CFG generated from an easy level code snippet is not flawless because GPT only use raw code for each node in the CFG. In the future work, researchers can design prompts to leverage GPT to generate other visualization graphs as shown in section II.D.

V. Conclusion and Future Work

By categorizing and examining the distribution of research across four pivotal areas—code comment generation, EEG-based cognitive studies, experimental comprehension studies, and code visualization techniques—this paper has provided a detailed overview of the methodologies and approaches employed to enhance the understanding of code among developers. Moreover, the exploration of LLMs, particularly in the realms of comment generation and code visualization, underscores the transformative potential these technologies hold for the future of improving code comprehension. In future work, we plan to refine and tune LLMs to enhance the accuracy of comment and visualization graphs other than CFG generation.

Acknowledgment

This research is supported by the NSF grant CCF-2342355.

References

[1] How long do software engineers stay at a job? https://www.linkedin.com/pulse/how-long-do-software-engineers-stay-job-firas-abbasi/

[2] Software Maintenance Costs in Brief: https://www.scnsoft.com/software-development/maintenance-and-support/costs

[3] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer—An investigation of how developers spend their time. In Proceedings of the IEEE 23rd International Conference on Program Comprehension. IEEE, 25–35.

[4] Wong, Edmund, Jinqiu Yang, and Lin Tan. "Autocomment: Mining question and answer sites for automatic comment generation." 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013.

[5] Liang, Yuding, and Kenny Zhu. "Automatic generation of text descriptive comments for code blocks." Proceedings of the AAAI conference on artificial intelligence. Vol. 32. No. 1. 2018.

[6] Loyola, Pablo, et al. "Content aware source code change description generation." Proceedings of the 11th International Conference on Natural Language Generation. 2018.

[7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, ''Deep code comment generation,'' in Proc. 26th Conf. Program Comprehension, May 2018, pp. 200–210.

[8] Gonçales, Lucian José, Kleinner Farias, Lucas Silveira Kupssinskü, and Matheus Segalotto. "An empirical evaluation of machine learning techniques to classify code comprehension based on EEG data." Expert Systems with Applications 203 (2022): 117354.

[9] Ishida, Toyomi, and Hidetake Uwano. "Synchronized analysis of eye movement and EEG during program comprehension." In 2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP), pp. 26-32. IEEE, 2019.

[10] Lin, Yu-Tzu, Yi-Zhi Liao, Xiao Hu, and Cheng-Chih Wu. "EEG activities during program comprehension: An exploration of cognition." IEEE Access 9 (2021): 120407-120421.

[11] Gonçales, Lucian Jose, Kleinner Farias, Lucas Kupssinskü, and Matheus Segalotto. "The effects of applying filters on EEG signals for classifying developers' code comprehension." Journal of applied research and technology 19, no. 6 (2021): 584-602.

[12] Swidan, Alaaeddin, and Felienne Hermans. "The effect of reading code aloud on comprehension: an empirical study with school students." In Proceedings of the ACM Conference on Global Computing Education, pp. 178-184. 2019.

[13] Börstler, Jürgen, and Barbara Paech. "The role of method chains and comments in software readability and comprehension—an experiment." IEEE Transactions on Software Engineering 42, no. 9 (2016): 886-898.

[14] Wyrich, Marvin, Justus Bogner, and Stefan Wagner. "40 years of designing code comprehension experiments: A systematic mapping study." ACM Computing Surveys 56, no. 4 (2023): 1-42.

[15] Cornelissen B, Zaidman A, van Deursen A. A controlled experiment for program comprehension through trace visualization[J]. IEEE Transactions on Software Engineering, 2010, 37(3): 341-355.

[16] Asenov D, Hilliges O, Müller P. The effect of richer visualizations on code comprehension[C]//Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems. 2016: 5040-5045.

[17] Yin R, Keller R K. Program comprehension by visualization in contexts[C] International Conference on Software Maintenance, 2002. Proceedings. IEEE, 2002: 332-341.

[18] Umphress D A, Hendrix T D, Cross Ii J H, et al. Software visualizations for improving and measuring the comprehensibility of source code[J]. Science of Computer Programming, 2006, 60(2): 121-133.

[19] ChatGPT: https://chat.openai.com/

[20] Liu, Zhe, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. "Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions." arXiv preprint arXiv:2310.15780 (2023).

[21] Song, Xiaotao, et al. "A survey of automatic generation of source code comments: Algorithms and techniques." IEEE Access 7 (2019).

[22] Khan, Junaed Younus, and Gias Uddin. "Automatic code documentation generation using gpt-3." In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-6. 2022.

[23] Agarwal, Abhaya, and Alon Lavie. "Meteor, m-bleu and m-ter: Evaluation metrics for high-correlation with human rankings of machine translation output." In Proceedings of the Third Workshop on Statistical Machine Translation, pp. 115-118. 2008.

[24] Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "Bleu: a method for automatic evaluation of machine translation." In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp. 311-318. 2002.

[25] PublicAccess https://github.com/Paul-Cui-Bugkiller/Code-Comprehension-Review-and-Large-Language-Models-Exploration

[26] Lu, Shuai, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement et al. "Codexglue: A machine learning benchmark dataset for code understanding and generation." arXiv preprint arXiv:2102.04664 (2021).

[27] GeeksforGeeks https://www.geeksforgeeks.org/

[28] "Capitalize 1st character of all words having at least K characters" in GeeksforGeeks, https://www.geeksforgeeks.org/capitalize-1st-character-of-all-words-having-at-least-k-characters/

[29] Wang, Jianxun, and Yixiang Chen. "A Review on Code Generation with LLMs: Application and Evaluation." 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI). IEEE, 2023.

[30] Conrad Czejdo and Sambit Bhattacharya, "Increasing, Accessibility of Language Models with Multi-stage Information Extraction," Advances in Information Technology, Vol.13, No. 2, pp. 181-185Apri1 2022.

[31] Herry Sujaini, Samuel Cahyawijaya, and Arif B. Putra, "Analysis of Language Model Role in improving Machine Translation Accuracy for Extremely Low Resource Languages," Journal of Advances in Information Technology, Vol. 14, No.5, pp.1073-1081, 2023