2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT) | 979-8-3503-7024-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICCCNT61001.2024.10724566

Learning To Code With Text-Bison-001:A Beginner-Friendly Explainer for Python, C, Java

Kaivaram Sivarama Krishna
Department of CSE
Amrita School of Computing
Bengaluru
Amrita Vishwa Vidyapeetham
India
sivaramak214@gmail.com

Bommala Deepak Kumar
Department. of CSE
Amrita School of Computing
Bengaluru
Amrita Vishwa Vidyapeetham
India
deepakbommala@gmail.com

Morumpalli Deekshith Reddy
Department of CSE
Amrita School of Computing
Bengaluru
Amrita Vishwa Vidyapeetham
India
deekshithreddy940@gmail.com

Bheema Sai Varun
Department of CSE
Amrita School of Computing
Bengaluru
Amrita Vishwa Vidyapeetham
India
saivarun2024@gmail.com

Meena Belwal
Department of CSE
Amrita School of Computing
Bengaluru
Amrita Vishwa Vidyapeeptham
India
b_meena@blr.amrita.edu

***Abstract*— This work aims to develop *a* Code comprehensive interface for the three programming languages JAVA, C, and Python integrated with error detection and code optimization. Users can give input codes of these language codes in the system or any algorithm and it will explain to the user in detail, The working of the code via language-oriented parsers and tailored analysis techniques. The system breaks apart the code, line by line unveiling the code's logic, flow of control, and major algorithms. By employing quick yet in-depth descriptions of the code users always get to see behind the scenes with easier debugging, reading, and learning. Talking of multi-language support, 'CODE EXPLAINER' becomes a good resource for developers, students, and teachers providing insight into programming concepts**

Keywords: ***Code Explainer, Generative AI, Prompt Engineering, Lexical Analysis, Syntax Analysis, Semantic Analysis.***

I. Introduction

With the complex nature of software development, proficiency becomes necessary in not only the most fluent but all the most different programming languages. 'CODE EXPLAINER' emerges as a robust solution to this challenge, offering a versatile platform to facilitate code comprehension across three prominent languages: Java, C, and Python. The number of programming languages is always the situation that developers need to study language constructions that are unfamiliar to them. To solve the above-mentioned difficulties, 'CODE EXPLAINER' introduces a user-friendly interface where every section of code can be inserted in the language of the preference and thereafter, it provides a detailed explanation of the meaning of that code. Specifying a language-parsing method and analysis techniques the tool performs a syntax-based code analysis, which generates logic, control flow, and core algorithms at line-by-line granularity. The 'CODE EXPLAINER' tool has two roles, It helps developers and other learners to grasp programming concepts and techniques faster which will also help to improve their productivity as well as create a more inclusive programming space. 'CODE EXPLAINER' explains the given tasks and examples in programming. It decomposes algorithms into fragments, provides an analysis of their intricacy in English, and offers improvements/advice. The first feature makes the explanations and logs specific to the user, based on the coding history and experience while the second one makes the system come up with advice concerning the coding style of the user. Currently, it employs Large Language Models (LLMs) to help the users in enhancing the code and foreseeing the possible outcomes.

The tool includes Python, C, and Java depending on aspects such as ease of use, existence of paradigms, relevance to busine ss, availability of resources, syntax difference, variety of tasks, feasibility of solutions, flexibility, development of tools, and research domain. Python is used in CS 100 and C is used in CS106 while C and Java are low-level, compiled languages required in the software industry. The tool must be geared to Python and C only before expanding on other languages in the subsequent versions.

The key contributions of this work are:

1. Explanation of code in layman's terms.
2. Code optimization.
3. Error analysis and correction.

In Section 2, a brief of related work is mentioned. In Section 3, A detailed explanation of the complete workflow is discussed. The results are presented in Section 4. Section 5 concludes the work. Finally, Section 6 talks about the Future Scope.

II. Literature Survey

The work by Sarsa Sami et al. [1] and Chen et al. [2] mainly focuses on writing and evaluating a Visual Studio Code

extension known as GPTutor that uses the ChatGPT API to present automatic individualized explanations of pieces of code.
The experiments by T. Nakamura et al. [3] and Widyasari et al.[4] revealed that it delves into the integration of LLMs or ChatGPT in the code reviews with the help of the tailored explanations that may be generated within the process thus increasing its effectiveness.

The work by Linchen et al. [5] mainly focuses on the BASTS method to enhance the quality of automatic code summarization. However, traditional methods solely utilize Abstract Syntax Trees. BASTS works by cutting up code according to the dominator tree blocks of the Control Flow Graph to generate a split AST for each split of the code.

The work by Chen, Mark et al. [6] describes Codex which is a language model architecture trained on GitHub code snippets, references, crucial research on artificial language model development, and few-shot learning. It maintains ethical considerations as well as bias assessment, quoting much research provided about the dangers of NLP and biases.

The survey by Mialon et al. [7] on Augmented Language Models (ALMs) probes the advances in natural language processing by endowing language models with reasoning capabilities. It reduces the gap between compositionality in language models and to limitations of language models on arithmetic and symbolic induction.

Another survey by Ye Qinghao et al. [8] on the model has parallel expectations in both images and texts hence preserving the LM's generation while upping understanding of visuals. The experiments give evidence of instant execution, the ability to analyze, and in some instances, even advanced features like recognizing written scenes and correlating among multiple images.

The article by Widjojo et al. [9] explains a program that addressed 100 compiler error messages from numerous sources to help programmers learn strategies for effectively resolving compilation errors. Error messages from the compilers were correctly understood by GPT-4 while the paraphrases, such as "How to fix", proved to be slightly more helpful in prompting the model.

The survey by Taylor et al. [10] briefs the instruction of the DCC--help command, which supports beginners in programming courses that are often marked with a high failing rate. It provides contextual information which leads to generating actionable explanations such as compile-time and run-time errors, it has 90% accuracy in approximating correctness.

The experiments by Zhuo et al. [11] are compared to the full-parameter fine-tuning (FFT). Prominently LoRA many times gives the best trade-off of efficiency and cost- effectiveness. Big models show better performance but are still insecure and vulnerable to attacks. The study compares multiple fine-tuning methods offering insight into the set of trade-offs for LLMs

The survey by Barve et al. [12] discusses the work in areas of parallel parsing, compiler techniques, and multi-core architecture research. Conceptual footprints comprise AREA, which is a parallel object code and optimization of FORTRAN DO- loops. Speedup analysis, OOP (object-oriented parsing), and other avenues are discussed for their capacity-building abilities in parallelizing parsing.

The work by M. Kuznetsov et al. [13] explores syntax parsing factors that help in error detection in programming languages and how they compare with token parsing generators such as Yacc and Bison. The parser combinators have an exponential speed, this makes the complexity of operation polynomial O(n) which is compared to exponential O(n^2) for parser generators which is used for error detection.

The review of Kumar et al.[14], explains the parallel lexical analyzer to speed up lexical tasks in multi-core processors which would be applied on C/C++ source code. Moreover, the study proposes a delicate mechanism for detecting malicious URLs and classifying them into good or malicious classes. with great accuracy of Random Forest and Extra Tree Classifier algorithms.

The survey by S. Purve, et al. [15] briefs about the major leap in Formal Languages and Automata theory that comes with PEG and PP. These both provide a reasonable alternative to Context-Free Grammar which allows a backtracking parser approach with recursive descent. These difficulties are tackled within the grid method through optimized memory utilization and it employs memorization and a minimal amount of backtracking.

The experiments by R. Tiwang et al. [16], explain how the work is divided into two parts. In the first stage, STRANGE stands for an investigation platform being promised to improve the language understanding capabilities for the identification of plagiarism of code corresponding to existing products like JPlag. In the second stage, surface-level changes are identified and the detecting process is improved.

The work by J. Zhu et al. [17], tried to improve the processing of natural language descriptions in code generation by exposing the current dataset deficiencies and collecting more data. Analysis of code brought forth from fresh data demonstrated shortcomings of programmer tasks when compared to suitable code-writing techniques.

The survey by K. Dorofeev et al. [18] introduced general device configuration interfaces in different systems, which enable flexible symphonizes and error recovery, good for Industry 4.0 requirements. Evaluation through the industrial shows approach is a good choice because it can handle changes in behavioral models, programming languages, and runtime and this is done to promote service-oriented architecture (SOA) in manufacturing.

The work by E.Dehaerne et al. [19] provides the documentation, on refactoring(changes) which will be analyzed, and three paradigm shifts that happened in the process are also highlighted. RNNs, transformers, and CNNs are known ML models to be considered with subwords as

another technique to overcome vocabulary obstacles. Concerns incorporate processing massive amounts of data and guaranteeing data quality, specifically advancing the investigations concentrating on language model efficiency and abstract syntax tree representation.

The work by Vishvadas et al. [20] gives a review of an in-depth exploration of code obfuscation approaches that mainly tells about the essential objective of protecting developer's intellectual work while upholding high standards on software security.

The work by Likhith et al. [21], is based on prior research of natural language process and compiler design. It includes investigating existing methods of English sentence-to-code transfers mathematical operations are explained too. The goal of writing their compiler is scattered around the development of the interdisciplinary field and the opening issues being faced.

The work by Veerappan et al. [22], presents a rule-based Kannada morphological analyzer and generator implemented using finite state transducers. It uses them to expand on preliminary findings largely in the field of computational linguistics to create a custom designed for Kannada morphology.

The work by Nair et al. [23], gives a systematic review of morphological analyzers for Indian languages. This works to identify the current state of work in their area of study. The authors Krishna, ER et al. [24], applied generative AI for converting numerical code into mathematical functions.

The work in the direction of code explanation is limited. The proposed 'CODE EXPLAINER' is capable of providing full code explanation with the help of generative AI, prompt engineering, and also backend processes for many programming languages like Python, C, and Java.

III. Background & Proposed Methodology

The core database capabilities of MongoDB are This section dives deep into the working of the various components used and needed to develop the 'CODE EXPLAINER'. Further, The workflow of the 'CODE EXPLAINER' is depicted in Figure 1 and explained in this section.

A. Generative AI and its integration

Generative – AI stands for a type of AI that is capable of producing a text by cue provided or on an assigned subject. It is learned and uses the probability-based methods to arrive at values that look like a sample of the inputs and sometimes even trained looks have been applied. Generative AIs, for instance, non-trivial tasks such as the digital creation of images, and documents, as well as other natural language processes, are the tasks included in creative assignments. Google's specially developed generative AI which is bilingual 300-word explanations receiving 100% human explanations along with the word-to-word explanation. Text-Bison 001 model is used to generate the text type outputs only

B. Prompt engineering

Gen-Ai and LLMs operate like functions or methods that receive input systems similar to programming languages. The code explanatory tool utilizes prompts as input with the ability to dynamically adjust for the given programming language and provided code snippet. Such prompts help the 'CODE EXPLAINER' reader to process information, correct errors, and provide instruction in a sophisticated manner. This process contains a parameter called 'Temperature' which regulates the outputs based on the value provided by the developer.

C. API Integration

API integration enables seamless communication between the tool and the generative- AI service, allowing for on-demand text generation based on user inputs. The API key provided during configuration authenticates the tool's access to the Generative-Ai service, ensuring secure and authorized usage

D. Backend Processing

The backend processing pipeline for code explanation initiates upon receiving a code snippet as input. this pipeline is designed to analyze and comprehend the code, enabling the generation of coherent explanations. the first step in this pipeline is input validation, reminiscent of the initial stages of a traditional compiler's operation. here, the code snippet undergoes a rigorous process through a series of stages kind including lexical analysis, syntax analysis, and semantic analysis.

1. Lexical analysis

A lexical analyzer identifies the original language of the source code and converts it to a sequence that can be scanned and comprises constant segments, typographic instructions, operators, and literary text. The symbolic or the units alongside the tokens and regular expressions will be identified and the succession of a subsequent analysis will be laid out. In one of the languages, the most appropriate data structure is needed, and the method of algorithmic efficiency is selected. One will find that the usage of regular expressions is applied as suitable matches for search patterns.

2. Syntax analysis

Syntax Analysis is a term frequently used for parsing, where a parser or parser generator "extracts" the structure or syntax of the code snippet to make sure that it follows the rules and grammar of the selected programming language. The compiler similar to a regular one, tackles the accuracy of the syntax of the input code snippet to correct the syntactical errors. These levels involve actions like parsing algorithms whilst constructing a parse tree which represents the syntactic structure of the program.

3. Semantic analysis

This is the analysis level where analysis peels more into the code's meaning and consequence of the snippet other than the syntax used. the semantic analysis happens at this stage; it entails the review of code such as declared variables, function calls, and type compatibility.

4. Code Optimization

Modern compilers include code optimization techniques to improve the efficiency and performance of the code. In code explanation, this optimization is mainly based on a prompt given at the back end and this is also helpful to optimize the explanations, taking the base as the code inputs.

5. Interpreting prompts

As explained before, the prompts are the values of the functions of 'CODE EXPLAINER', these prompts encapsulate the task of explaining the code snippet and guide the model toward generating human-understandable explanations. these prompts are essential for producing relevant and informative explanations.

6. Generating Explanations

The main abstract player of this section is Generative AI, it generates explanations using Natural language that encapsulates the semantics, functionally, and key topics of the input code snippet. This process mirrors the translation of input code into machine-readable instructions by a compiler. the model produces explanations that convey the underlying logic and behavior encoded within the code snippet.

7. User interface

The front-end interface, built using Gradio, provides users with an easily accessible platform for interacting with the code explanation tool. Users can input code snippets by selecting the programming language and receive detailed explanations generated by the backend AI model. The interface enhances user experience and accessibility, making the code explanation tool accessible to upcoming developers of basic-level coding.

The complete workflow of the explainer is depicted in Figure1

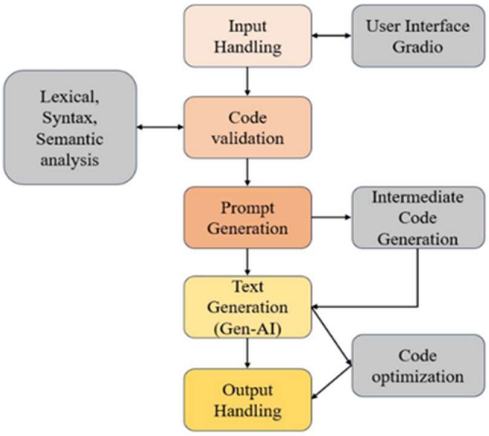

Fig.1. Work Flow of "CODE EXPLAINER"

The 'CODE EXPLAINER' pipeline in Figure 1 is initiated by input code analysis of the program using Gradio for lexical analysis, syntax analysis, and semantic analysis. The back-end itself then produces intermediate code and with the help of Text Bison 001 prepares an explanation that is displayed. During output management, the system enhances the quality of the code, also correcting errors that had been made, and this leads to the optimization of the code.

IV. Results and Discussions

This section explains the results from Google's Gemini API, the large language model (LLM), and the workings of the text generation model. First Accessing Google's Gemini API in

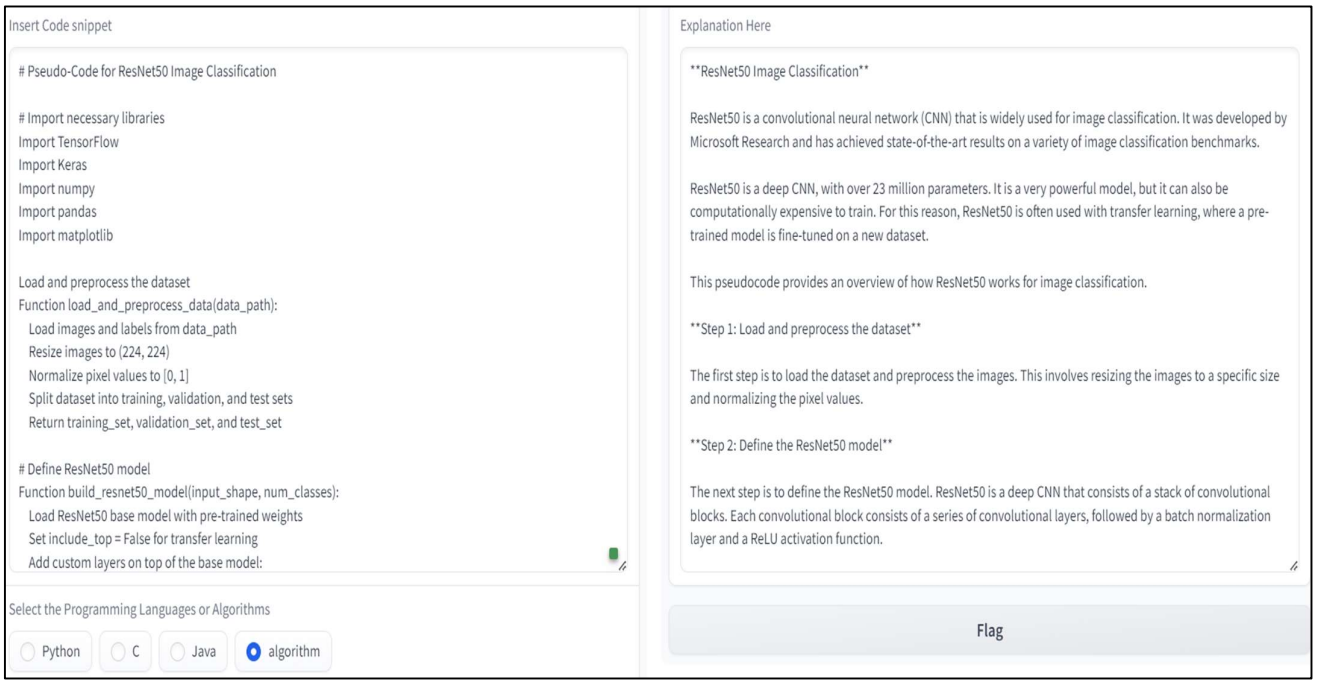

Fig. 2. Explanation of a Complex pseudo code DL Algorithm

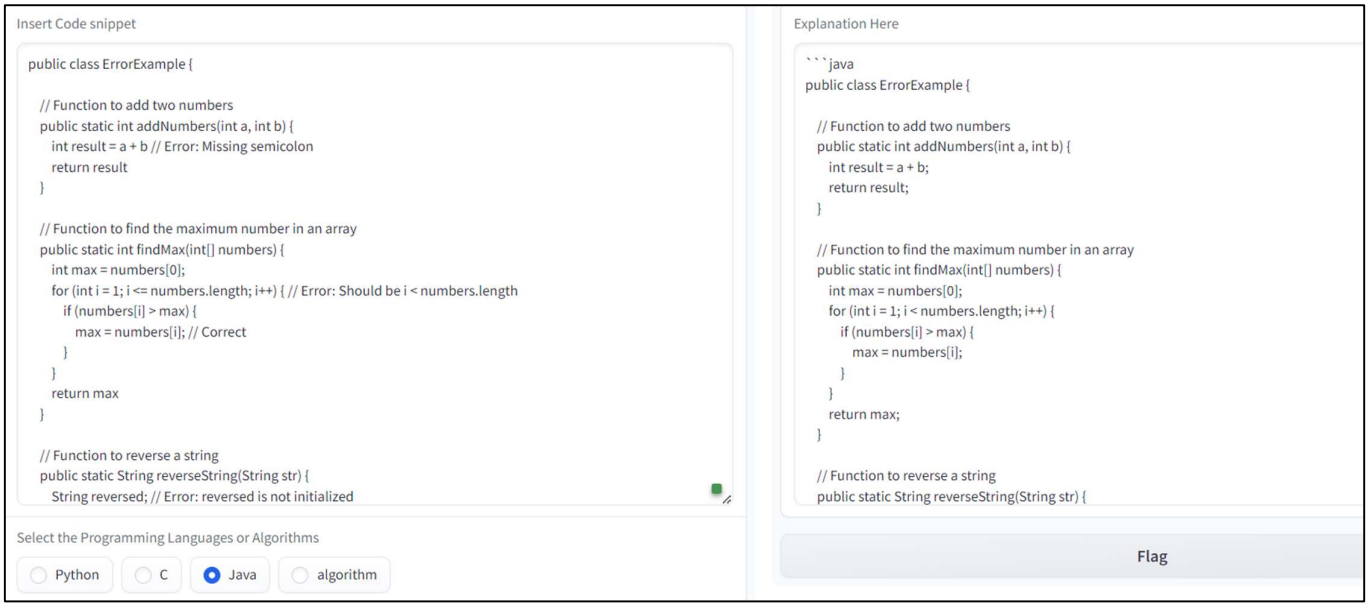

Fig.3. Error handling and correction of the Java code

the form of a key helps to access the Large Language model called PaLM, From this large language model PaLM a text generation model "text-bison 001" is used to generate only the text type of output. Gradio is used to create an interactive web interface that holds all the prompting parts as the backend. The Interface is named 'CODE EXPLAINER'. This is designed to explain the code in Layman's terms, which is very useful for beginners who are starting to learn codes. It takes the code as input and explains it in terms of modern compiler design. First, the code is tokenized then syntax and semantic analysis are performed, these are the backend parts that one cannot see, this is all done by giving simple prompts in the backend code.

In the next step, the users proceed through filling in the code accompanied by the preferred programming language. Clicking on submit. Next check whether it follows the instructions or not Further to this, the sample of the project is an example of code purpose, where the connection and location of errors are handled in addition to the rest of the code. The corrected version is given and the time and space complexity are calculated for optimizing code. The corrected version is given and the time and space complexity are calculated for optimizing.

The on-screen presentation of the proposed 'CODE EXPLAINER' is divided into two parts; The first block which is on the left side is the input block that takes the input from the user and the second block which is on the right side is the output block which displays the explanation of the given input, as shown in Figure 2 and Figure 3.

Figure 2 demonstrates the ability of 'CODE EXPLAINER' as the complex DL code is well explained. Similarly, Figure 3 displays the example of Error handling and correction of the Java code.

However, the 'CODE EXPLAINER' has been thoroughly tested for Error handling and correction in the languages: Python, C, and Java. The error-handling process deals with errors and produces corrected codes. This also includes the complexity analysis. Further the 'CODE EXPLAINER' is also able to explain the algorithms of Web Development. Thus, the user will learn different types of errors so that the user can benefit from them, but overall, the user will also expand their area of expertise.

V. Conclusion

Finally, 'Learning to code with textbison-001' uses cutting-edge language modeling and an interactive interface to offer a faster way to learn to code for beginner students. Through the integration of the Gemini API provided by Google and the PaLM language model, The 'CODE EXPLAINER' system provides code interpretations as well as goes through error detection and code optimization. In grades, the application distinguishes itself by the demonstration of core concepts of arrays, linked lists, various algorithms, other data structures, and error analysis in various choice of programming languages in this case JAVA, C, and Python. This project therefore roles out as an invaluable instrument to beginners, putting forward obviously comprehensible and well- well-systematized explanations that, themselves, simulate the scenarios that the learners can practically practice on.

VI. Future Scope

The "CODE EXPLAINER" might apply complex code analysis procedures and even use artificial intelligence to analyze such code, not to mention, expand its language capabilities and include other languages like Python and C; Java. It may also involve linking this to an IDE, explaining the operations in a different language, and Code generation.

Future research directions must be focused on ethical viewpoints and education outcomes.

REFERENCES

[1] Sarsa, Sami, Paul Denny, Arto Hellas, and Juho Leinonen. "Automatic generation of programming exercises and code explanations using large language models." In Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1, pp. 27-43.2022

[2] Chen, Eason, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. "GPTutor: a ChatGPT-powered programming tool for code explanation." In International Conference on Artificial Intelligence in Education, pp. 321-327. Cham: Springer Nature Switzerland, 2023.

[3] T. Nakamura, A. Monden, M. Sasakura and H. Uwano, "Effectivenessof Explaining a Program to Others in Finding Its Bugs," 2021 IEEE/ACIS 22nd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing(SNPD), Taichung, Taiwan, 2021, pp. 248-253, doi: 10.1109/SNPD51163.2021.9704932

[4] Widyasari, Ratnadira, Ting Zhang, Abir Bouraffa, and David Lo. "Explaining Explanation: An Empirical Study on Explanation in CodeReviews." arXiv preprint arXiv:2311.09020 (2023).

[5] Lin Chen, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li,and Rongxin Wu. "Improving code summarization with block-wise abstract syntax tree splitting." In 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pp. 184-195. IEEE, 2021.

[6] Chen, Mark, et al. "Evaluating large language models trained on code."arXiv preprint arXiv:2107.03374 (2021).

[7] Mialon, Grégoire, et al. "Augmented language models: a survey." arXiv preprint arXiv:2302.07842 (2023).

[8] Ye, Qinghao, et al. "plug-owl: Modularization empowers large language models with multimodality." arXiv preprint arXiv:2304.14178 (2023).

[9] Widjojo, Patricia, and Christoph Treude. "Addressing compiler errors:Stack overflow or large language models?." arXiv preprint arXiv:2307.10793 (2023).

[10] Taylor, Andrew, et al. "Dcc--help: Transforming the Role of the Compiler by Generating Context-Aware Error Explanations with LargeLanguage Models." Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. 2024.

[11] Zhuo, Terry Yue, et al. "Astraios: Parameter-Efficient Instruction Tuning Code Large Language Models." arXiv preprint arXiv:2401.00788 (2024).

[12] Barve and B. K. Joshi, "Parallel syntax analysis on multi-core machines," 2014 International Conference on Parallel, Distributed and Grid Computing, Solan, India, 2014, pp. 209-213, doi: 10.1109/PDGC.2014.7030743.

[13] M. Kuznetsov and G. Firsov, "Syntax Error Search Using Parser Combinators," 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), St. Petersburg, Moscow, Russia, 2021, pp. 490-493, doi: 10.1109/ElConRus51938.2021.9396311.

[14] Kumar and S. Maity, "Detecting Malicious URLs using Lexical Analysis and Network Activities," 2022 4th International Conference on Inventive Research in Computing Applications (ICIRCA), Coimbatore, India, 2022, pp. 570-575, doi: 10.1109/ICIRCA54612.2022.9985586.

[15] S. Purve, P. Gogte and R. Bhave, "Stack and Queue Based Parser Approach for Parsing Expression Grammar," 2023 11th International Conference on Emerging Trends in Engineering & Technology - Signal and Information Processing (ICETET - SIP), Nagpur, India, 2023, pp. 1-6, doi: 10.1109/ICETET-SIP58143.2023.10151539

[16] R. Tiwang, T. Oladunni and W. Xu, "A Deep Learning Model for Source Code Generation," 2019 Southeast Con, Huntsville, AL, USA, 2019, pp. 1-7, doi: 10.1109/SoutheastCon42311.2019.9020360.

[17] J. Zhu and M. Shen, "Research on Deep Learning Based Code Generation from Natural Language Description," 2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), Chengdu, China, 2020, pp. 188-193, doi: 10.1109/ICCCBDA49378.2020.9095560.

[18] K. Dorofeev, S. Bergemann, T. Terzimehić, J. Grothoff, M. Thies and A. Zoitl, "Generation of the Orchestrator Code for Skill-Based Automation Systems," 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Vasteras, Sweden, 2021, pp. 1-8, doi: 10.1109/ETFA45728.2021.9613728.

[19] E. Dehaerne, B. Dey, S. Halder, S. De Gendt and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," in IEEE Access, vol. 10, pp. 82434-82455, 2022, doi: 10.1109/ACCESS.2022.3196347 Image Processing and Capsule Networks, pp. 491-509. Singapore: Springer Nature Singapore, 2023.

[20] Vishwas, Gade, Nayini Sai Nithin, Peddineni Varshith, and Meena Belwal. "Unveiling the World of Code Obfuscation: A Comprehensive Survey." In 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-8. IEEE, 2023.

[21] Likhith, Arlagadda Naga, Kothuru Gurunadh, Vimal Chinthapalli, and Meena Belwal. "Compiler For Mathematical Operations Using English Like Sentences." In 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-6. IEEE, 2023.

[22] Veerappan, Ramasamy, et al. "A rule-based Kannada morphological analyzer and generator using finite state transducer."International Journal of Computer Applications 27.10 (2011): 45-52.

[23] Nair, J., Aiswarya, L.S., Sruthy, P.R. (2021). A Study on Morphological Analyser for Indian Languages: A Literature Perspective. In: Singh, M., Tyagi, V., Gupta, P.K., Flusser, J., Ören, T., Sonawane, V.R. (eds) Advances in Computing and Data Sciences. ICACDS 2021. Communications in Computer and Information Science, vol 1440. Springer, Cham. https://doi.org/10.1007/978-3-030-81462-5_11.

[24] Krishna, ER Adwaith, Akhbar Sha, Kalwa Anvesh, Nancharla Abhinay Reddy, Bhupathi Shwejan Raj, and K. S. Nisha. "Generative AI-Driven Approach to Converting Numerical Code into Mathematical Functions." In *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*, pp. 661-666. IEEE, 2023.