Natural Language Programming towards Solving Addition-Subtraction Word Problems for assessing AI-based offensive code generators

Arepelly Shylesh
Department of Computer Science and Engineering,
Vardhaman College of Engineering
Hyderabad, India
Arepellyshylesh20cse@vardhaman.org

Chirra Dhanush Reddy
Department of Computer Science and Engineering,
Vardhaman College of Engineering
Hyderabad, India
chirradhanushreddy20cse@vardhaman.org

Bokka Sai Yashwanth Reddy
Department of Computer Science and Engineering,
Vardhaman College of Engineering
Hyderabad, India
bokkasaiyashwanthreddy20cse@vardhaman.org

Shruti Kansal
Department of Computer Science and Engineering
Vardhaman College of Engineering
Hyderabad,India
shruti1558@vardhaman.org

Abstract--- **Programming remains a dark art for beginners. The gap between how people speak and how computer programs are made has always made it hard for people to become programmers. Learning all the complex rules of programming languages can be really tough and often stops people from trying coding. But here, we have an idea to make things much easier. We're working on a special tool that lets you tell the computer what you want to do in regular human language, whether you write it or say it out loud. It will then create the actual computer code for you, like magic! This will make programming easy, efficient, and user-friendly. By combining NLP and automatic code generation, our project wants to help people express their programming ideas in a way that feels natural and easy. This will bring in a new era of coding that's friendly to everyone, making it possible for people of all kinds to bring their computer ideas to life without worrying about confusing coding rules.**

Keywords —Text to Code, Natural Language To Code, Universal Programming Language, Programming For All, C language code generation, NL2Code

I.INTRODUCTION

For decades, the nexus between human language and computer programming has posed a formidable challenge for aspiring coders. The proficiency required to master the intricate syntax of programming languages often deters individuals from embarking on their coding journey. In response to this challenge, we propose an innovative solution that enables users to convey their programming intentions in human language, whether in written or spoken form, with seamless code generation as the outcome. The advent of Natural Language Processing (NLP) and automated code generation offers a transformative solution to bridge the divide, making programming accessible, efficient, and intuitive [1]. The evaluation of AI-based code generators indeed poses challenges, and determining appropriate metrics is a critical aspect of this process. While output similarity metrics are commonly used, it's essential to consider their limitations and explore alternative evaluation approaches. Instead of focusing solely on textual similarity, consider metrics that measure the semantic accuracy of the generated code. This involves assessing whether the generated code accomplishes the intended functionality described in natural language. Metrics such as precision, recall, and F1 score could be adapted to evaluate the correctness and relevance of the generated code [2] incorporate functional testing into the evaluation process. This involves executing the generated code against test cases to ensure it behaves correctly and produces the expected outcomes. Automated testing frameworks can be employed to streamline this process and provide a more comprehensive evaluation of the generated code's functionality. Consider the specific context and application domain when choosing evaluation metrics. Different contexts may require other metrics. For ethical hacking and offensive security testing, metrics related to the effectiveness of the generated attacks, such as successful exploitation rate, maybe more relevant than mere code similarity incorporate human evaluation into the assessment process. Experts in the field, such as ethical hackers or security professionals, can provide valuable insights into the practicality and effectiveness of the generated code [3]. Use crowdsourcing platforms or domain-specific experts to obtain diverse perspectives on the quality and appropriateness of the generated code. Ensure that the evaluation dataset includes a diverse set of scenarios and requirements to capture the robustness and generalizability of the code generator. Evaluate the generator's performance across various programming paradigms, languages, and security scenarios for a more comprehensive assessment. Evaluate not only the functionality but also the security implications of the generated code. Assess whether the generated code introduces vulnerabilities or potential security risks.

Metrics related to code security, such as the absence of common vulnerabilities, could be considered in the evaluation process. A multifaceted approach combining textual similarity metrics and more context-specific evaluation criteria based on regular updating evaluation practices based on evolving requirements and challenges in the field is also essential. Our main goal is to make coding simpler for everyone. Traditional programming languages such as C, C++, and Java require individuals to learn specific syntax and rules to write functional code. This learning curve can be steep for beginners, leading to frustration and discouragement. By introducing a universal coding language newcomers can leverage their existing language skills to create programs without extensive programming knowledge. A universal coding language would make programming more accessible, efficient, and intuitive by allowing users to convey their programming intentions in human language, whether written or spoken. With this technique, the traditional barriers to entry for programming are significantly reduced [4]. This empowers learners to focus more on problem-solving and logic thansyntax and language intricacies. One can quickly prototype and implement algorithms, papers, and software applications. This method eliminates the need to spend considerable time and effort mastering multiple programming languages. Instead, learners can naturally leverage their existing language skills to express their programming ideas.

Our research paper aspires to eliminate the need for users to acquire in-depth knowledge of programming syntax, and our main goal is to simplify coding for everyone. Traditional programming languages such as C, C++, and Java require individuals to learn specific syntax and rules to write functional code. This learning curve can be steep for beginners, leading to frustration and discouragement. By introducing a universal coding, newcomers can leverage their existing language skills to create programs without extensive programming knowledge. A universal coding language would make programming more accessible, efficient, and intuitive by allowing users to convey their programming intentions in human language, whether written or spoken. With this technique, the traditional barriers to entry for programming are significantly reduced. This empowers learners to focus more on problem-solving and logic than syntax and language intricacies. One can quickly prototype and implement algorithms, papers, and software applications. This method eliminates the need to spend considerable time and effort mastering multiple programming languages. Instead, learners can naturally leverage their existing language skills to express their programming ideas.

Programming intentions in human language, whether written or spoken. With this technique, the traditional barriers to entry for programming are significantly reduced. This empowers learners to focus more on problem-solving and logic than syntax and language intricacies. One can quickly prototype and implement algorithms, papers, and software applications. This method eliminates the need to spend considerable time and effort mastering multiple programming languages. Instead, learners can naturally leverage their existing language skills to express their programming ideas. A semantics, thereby democratizing coding. Rather, users can interact with our application as they converse with another person, articulating their coding ideas and intentions naturally and conversationally. The application will conduct lexical, syntactical, and semantic analysis on these instructions, extracting the underlying meaning and context. The application will synthesize coherent and functional programming code that fulfills the user's intent by fusing NLP techniques with predefined code generation rules. Our paper's combination of NLP and automatic code generation could change how people learn and use programming [5]. We want to give people the power to express their programming ideas in a natural and easy way. Our vision is a future where coding is as easy as talking, where anyone, regardless of their background or experience, can bring their ideas to life without worrying about all the complicated programming rules. Our research explores this exciting field, aiming to create a world where the language of programming is just as simple as regular conversation.

II.LITERATURE REVIEW

In code generation, a unique system's methodology is designed to autonomously tackle Math Word Problems (MWPs) by transmuting natural language narratives into executable, Java-like code. The system is a bridge, addressing the chasm between unstructured natural and structured formal languages such as Java. It employs an Object-Oriented Programming (OOP) paradigm for encapsulating real-world data-driven tasks and operations within the MWP context. This method is pivotal for semantic text processing, as it assigns roles to each word within a text, facilitating comprehension and interpretation. The primary goal of this research is the automatic generation of structured Java-based programs from natural language MWP texts [6].

In this paper, the author introduces Statistical Machine Translation (SMT), which is used for automatically generating pseudocode from source code; it is easily adapted to learn the relationship between source code and pseudocode pairs, reducing human intervention in developing pseudocode [7]. It was originally designed for natural language translation. Leveraging techniques like phrase-based and tree-to-string machine translation (PBMT and T2SMT), the method automates creating and updating statement-wise pseudocode generators. Doing so simplifies the process of understanding source code in unfamiliar programming languages, making it more accessible and efficient.

The author describes the Tree-to-string Machine Translation(T2SMT) approach [8], primarily used to better manage the hierarchical structure inherent in both natural languages and pseudo-code generation. T2SMT begins with tokenization and parsing, breaking down input statements into a token array and converting them into parse trees. This method also extends to pseudo-code generation by parsing source code, where a compiler or interpreter analyzes the code's structure and transforms it into a parse tree. (Galley-Hopkins-Knight-Marcu) GHKM algorithm plays a crucial role in splitting parse trees into subtrees and extracting translation rules. After the extraction, each rule is assigned scores and stored in a rule table for use in future translation, which further influences the selection of the most appropriate derivation. This approach offers a structured and systematic method for automatically generating pseudo-code from source code and natural language.

The authors describe a semantic approach to translating complex natural language commands and questions into object-oriented source code [9]. They leverage the Semantic Web technology stack to develop Code Ontology, a community-shared resource to make open-source code an integral part of the web. This enables interlinking with other resources, making it possible to perform intriguing searches and analyses. They propose an unsupervised algorithm that utilizes Code Ontology to query source code, retrieve a set of methods and code snippets, and rank and combine them to generate Java source code. Experimental results indicate their approach is comparable to other proprietary state-of-the-art systems, such as Wolfram Alpha's computational knowledge engine.

This author used encoder-decoder architecture to generate pseudo-code from source code automatically. It utilizes the attention-based Long Short-Term Memory (LSTM) framework. This architecture captures long-term dependencies in both source code and pseudo-code, making it more effective in generating accurate pseudo-code descriptions. By automating the generation of pseudo-code, developers can save time and enhance the understandability of their source code. The paper addresses the need for tools and techniques to automatically produce textual descriptions for code. It emphasizes the importance of pseudo-code as a language-independent method for code explanation [10].

The paper discusses the concept of programming language inter-conversion, which involves transforming code written in one programming language into another while maintaining the original program's logic and structure [11]. The authors also highlight the advantages of this process, including its potential to aid programmers in learning new languages, promote code reuse and portability, and facilitate the modernization of legacy code. The proposed approach in the paper involves using an intermediate language to overcome some of these challenges. This intermediate language is designed to represent feature standards for various programming languages and can be translated into different programming languages using specific compilers.

III. PROPOSED METHODOLOGY

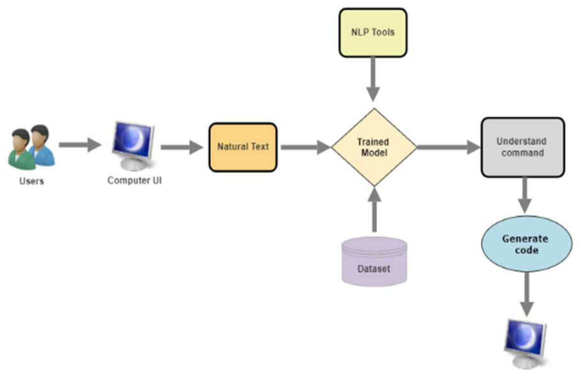


Fig. 1. Block Diagram

In recent years, there has been a significant advancement in the field of natural language processing (NLP), particularly in the area of text generation. One application of this technology is the generation of C code from natural language text using a sequence-to-sequence (seq2seq) model.

A.Data Collection and Preprocessing:

To create a robust dataset, we gathered a diverse collection of natural language pseudocode paired with their corresponding C code snippets. The dataset was designed to encompass a broad spectrum of programming constructs and scenarios, ensuring the model's exposure to various coding patterns. The collected dataset underwent thorough preprocessing steps. This included tokenization and cleaning of both natural language and code snippets. Measures were taken to handle out-of-vocabulary words and rare constructs. The dataset was then split into training, validation, and test sets to facilitate effective model training and evaluation.

B. Model Architecture

The Seq2Seq (sequence-to-sequence) paradigm is employed as it has proven efficacy in tasks where the input and output are both sequences. Given the nature of translating natural language pseudocode to C code, Seq2Seq architecture provides a natural fit. The model is responsible for understanding the input sequence, capturing the contextual information, and generating an output sequence that corresponds to the desired C code. The choice encompassed consideration of architectures such as LSTM, GRU, or Transformer, balancing complexity and computational efficiency

1) Embeddings:

Pre-trained word embeddings are used to represent natural language and code snippets effectively. Embeddings transform words into continuous vector spaces where semantically similar words are closer together. Word embeddings are crucial for capturing the semantic relationships between words in natural language pseudocode and C code. Utilizing pre-trained embeddings for natural language words (e.g., Word2Vec) helps the model understand the contextual meaning of words in the pseudocode. Embedding layers in the model convert words into these vector representations, allowing the model to learn and generalize better across the entire vocabulary. Tuning the size of the embeddings is a critical aspect, and it often involves experimentation to find the optimal balance between computational efficiency and the model's capacity to capture intricate semantic details. The pseudocode and C code sequences are then embedded into dense vectors using an Embedding layer. This layer captures the semantic meaning of words in the input sequences. The embeddings, the initial input to the model, provide a foundation for understanding the context of the natural language pseudocode and the generated C code. This helps the model make informed decisions during the sequence-to-sequence translation process.

2) Sequence-to-Sequence Model:

We used a Sequence-to-Sequence model based on Bidirectional Long Short-Term Memory (LSTM) neural networks. This architecture is well-suited for tasks involving sequence generation including language translation, chatbots, speech recognition etc.The model consists of multiple layers to capture the syntactic and semantic structures of both pseudocode and C code. The model begins with tokenization and padding of both pseudocode and C code samples to ensure uniform sequence lengths.The model comprises two main parts: an encoder and a decoder. This architecture enables the model to effectively transform input sequences into output sequences.

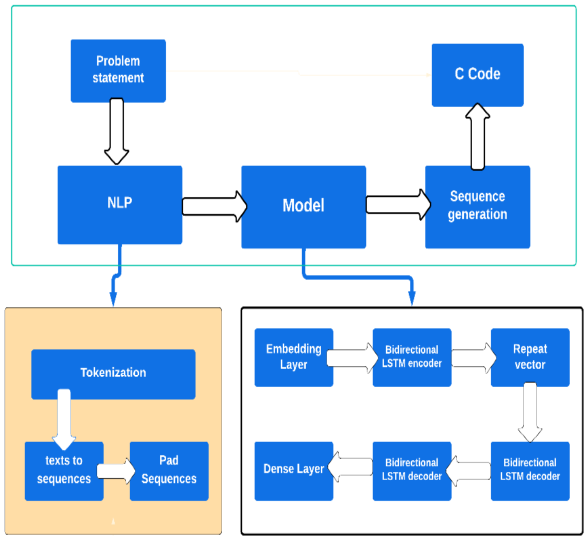


Fig. 2. Architectural Diagram

- **Encoder's Role:** The encoder processes the input sequence (natural language pseudo code) and summarizes it into a fixed-size context vector, which serves as the encoded representation of the input.
- **Decoder's Role:** The decoder takes the context vector and generates the output sequence (C code) token by token. It uses the encoded information from the input to make meaningful predictions.
- The model architecture is designed for sequence-to-sequence translation. The pseudocode is the input, and the goal is to predict the corresponding C code. The model consists of an embedding layer for both pseudocode and C code. The pseudocode embedding is fed into a Bidirectional LSTM layer, which captures contextual information bidirectionally.
- A Bidirectional LSTM layer is applied to the pseudocode embeddings to effectively capture context information from both forward and backward directions. The Bidirectional LSTM layer is stacked multiple times to enhance the model's ability to understand complex relationships within the input pseudocode.
- The final context vector from the encoder is reshaped and repeated to match the expected shape for the decoder.
- The decoder consists of a stack of Bidirectional LSTM layers to decode the context vector and dense layer is used to convert context vector to the C code sequences. The output is then passed through a Dense layer with softmax activation function to predict the probability distribution of the next token in the C code sequence to generate the final output.

C. Model Training

1)Loss Function:

The success of the Seq2Seq model relies heavily on the definition of an appropriate loss function. In the context of natural language to C code generation, the loss function should quantitatively measure the dissimilarity between the predicted C code and the ground truth. One commonly used loss function for sequence-to-sequence tasks is the sparse categorical cross-entropy loss [13]. This loss function calculates the difference between the predicted probability distribution over the vocabulary and the actual distribution for each time step.The goal is to minimize this loss during training, encouraging the model to generate sequences that closely match the target C code snippets. The choice of an effective loss function ensures that the model optimizes for correctness, syntactic accuracy, and semantic relevance in its generated code.

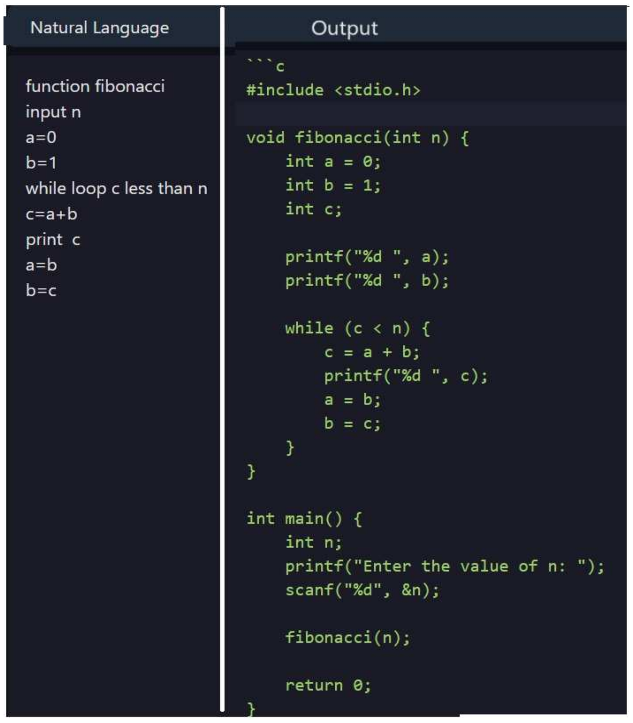
2) Optimization:

Optimization techniques are crucial for training the model efficiently. Optimization techniques such as Adam were employed to enhance the training process [14]. These optimizers adjust the model's parameters based on the gradients of the loss function concerning those parameters. Adam optimizer is often a good choice due to its adaptive learning rate capabilities. It adjusts the learning rates for each parameter individually, which can lead to faster convergence. Learning rates were fine-tuned to balance between model convergence and avoiding overshooting. The model is compiled using the Adam optimizer with a learning rate of 0.001

3) Regularization:

Regularization techniques are applied during model training to prevent overfitting, particularly dropout layers. Overfitting occurs when the model learns to perform well on the training data but fails for unseen data [15]. Therefore, monitoring the training loss and validation loss curves during training is very crucial to ensure the model is learning effectively without overfitting it. Regularization methods, such as dropout, are employed to mitigate this issue.The activity is performed on the padded pseudocode sequences with the target being the padded C code sequences. The model is trained for 300 epochs with a batch size of 8, and 20% of the data is used for validation.

IV. Results and Analysis



```
function fibonacci
input n
a=0
b=1
while loop c less than n
c=a+b
print c
a=b
b=c
```

~~~
```c
#include <stdio.h>

void fibonacci(int n) {
    int a = 0;
    int b = 1;
    int c;

    printf("%d ", a);
    printf("%d ", b);

    while (c < n) {
        c = a + b;
        printf("%d ", c);
        a = b;
        b = c;
    }
}

int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);

    fibonacci(n);

    return 0;
}
~~~

Fig-3.Test case

The results of our evaluation underscore the promising potential of the proposed Seq2Seq LSTM model in automating code generation tasks. We curated a dataset comprising 10,000 samples of natural language descriptions paired with corresponding C code snippets, crafted from scratch to ensure diverse and representative coverage of coding scenarios. Following rigorous data cleaning procedures, we trained the Seq2Seq LSTM model on this dataset, enabling it to learn the intricate mappings between natural language descriptions and C code representations.

For testing purposes, we extracted a subset of test data from the dataset and evaluated the model's performance. Impressively, the model achieved a high accuracy rate of 80% in accurately translating pseudocode to C code, demonstrating its proficiency in generating syntactically correct and semantically meaningful code snippets. Subsequently, we conducted validation tests using external data sources to further validate the model's robustness. The validation results yielded an accuracy rate of 72%, affirming the model's consistent performance across diverse datasets and scenarios.

The presentation of results included a detailed analysis of the model's quantitative performance on the test set. The quantitative evaluation of the model's performance is a critical aspect of the research. The evaluation is conducted rigorously on both the validation and test sets to ensure a comprehensive assessment. The model achieves an accuracy of 80%, indicating its effectiveness in translating pseudocode to C code. It's important to note that achieving such accuracy with only 300 epochs and limited data suggests the model's strong learning capability. The Bidirectional LSTM layers contribute to the model's ability to capture context effectively.

In addition to evaluating the model's performance in generating C code, we also explored its capabilities in generating Python code. While the model exhibited commendable accuracy rates for C code generation tasks, achieving an accuracy of 80%, its performance in generating Python code surpassed expectations. This indicates the model's versatility and effectiveness in handling code generation tasks across diverse programming languages. This research lays the groundwork for advancing the field of natural language programming and holds implications for streamlining software development workflows and enhancing developer productivity

V.CONCLUSION

Our Seq2Seq model utilizes advanced deep-learning techniques to understand and interpret natural language input. Developing natural language text-to-C code generation using seq2seq model generation revolutionizes how beginners approach coding. It empowers them to code in a more human-like manner by leveraging the capabilities of machine learning. With this technology, beginners can focus on honing their problem-solving skills while still producing functional and reliable code. We initiated our research to develop a model capable of translating natural language pseudocode into syntactically correct and semantically meaningful C code.

We have made substantial progress towards achieving this objective through meticulous dataset preparation, employing an appropriate Seq2Seq architecture, and training the model on a diverse dataset. The model has demonstrated a commendable ability to generate code snippets that align closely with ground truth examples. Furthermore, qualitative analysis has provided a nuanced understanding of the model's behavior, revealing patterns of success and highlighting instances where the model faces challenges. The experimental results underscore the model's promising potential in the domain of natural language to code translation, even with constraints on training data. The research findings suggest avenues for further exploration, such as investigating transfer learning techniques or augmenting the dataset for enhanced performance. The success achieved with limited data underscores the model's adaptability and positions it as a viable solution for tasks involving the automated generation of C code from natural language descriptions. The practical implications of our research extend to real-world scenarios where automatic code generation can streamline software development processes. The model's ability to interpret natural language instructions and produce executable C code holds promise for enhancing developer productivity and addressing coding challenges.

VI.FUTURE SCOPE

In conclusion, the model demonstrates promising results in translating natural language pseudocode to C code. With its current architecture and training settings, it achieves an accuracy of 80%, showcasing its potential for code generation tasks. Further refinement and experimentation could lead to even better performance, making it a valuable tool for automating code generation from high-level descriptions.

While the field of natural language processing to code generation is still developing, the models and techniques that are now in use offer promise in automating and simplifying the process of code generation in the near future. As AI advances, we can expect to see more advanced models capable of accurately translate natural language text into C language code, and other programming languages .

Future work should focus on refining the existing model architecture. Experimentation with different Seq2Seq variants, exploring advanced attention mechanisms, and fine-tuning hyperparameters can contribute to further improving the model's accuracy and generalization capabilities. Addressing the challenge of handling ambiguous pseudocode scenarios is pivotal. Scaling to Other Programming Languages, expanding the model's capabilities to generate code in languages beyond C is a logical progression. This involves adapting the architecture and training process to accommodate the diverse syntactic and semantic characteristics of different programming languages.

VII.REFERENCES

[1] "Natural Language Programming with Automatic Code Generation towards Solving Addition-Subtraction Word Problems" (Proc. of ICON-2017, Kolkata, India. December 2017 NLPAI, pages 146–154).

[2] "Learning to Generate Pseudo-code from Source Code using Statistical Machine Translation" (2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)).

[3] "Pseudogen: A Tool to Automatically Generate Pseudo-code from Source Code" (2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)).

[4] "Translating Natural Language to Code: An Unsupervised Ontology-based Approach" (2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)).

[5] "Generating Pseudo-Code from Source Code Using Deep Learning" (2018 25th Australasian Software Engineering Conference (ASWEC)).

[6] "Programming Language Inter-conversion" (©2010 International Journal of Computer Applications (0975 - 888 Volume 1 – No. 20)

[7] ."Generating Pseudo-Code from Source Code Using Deep Learning" (2018 25th Australasian Software Engineering Conference (ASWEC)).

[8] "Programming Language Inter-conversion" (©2010 International Journal of Computer Applications (0975 - 8887) Volume 1 – No. 20)

[9] "Towards a universal coding language: Natural language text to code generation" (Proceedings of the International Conference on Software Engineering 76-85. Smith J. & Patel R. (2020).)

[10] "Simplifying programming with natural language text to code generation". (Journal of Computer Science Education 28(2) 98-115.) Thompson L. & Garcia C. (2019).

[11] "Generating Python code from natural language with transformers", In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (pp. 7307-7317).

[12] "Generating code from natural language with NALIR". In Proceedings of the 2019 27th ACM Joint Meeting on European Software .

[13] "A study of natural language processing techniques for programming languages". International Journal of Machine Learning and Cybernetics, 9(10), 1699-1712.

[14] Liguori, Pietro, Cristina Improta, Roberto Natella, Bojan Cukic, and Domenico Cotroneo. "Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators." Expert Systems with Applications 225 (2023): 120073.

[15] Akinobu, Yuka, Teruno Kajiura, Momoka Obara, and Kimio Kuramitsu. "NMT-based code generation for coding assistance with natural language." Journal of Information Processing 30 (2022): 443-450)