



e-ISSN: 2319-8753 | p-ISSN: 2347-6710

# IJIRSET

International Journal of Innovative Research in  
**SCIENCE | ENGINEERING | TECHNOLOGY**



# INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

Volume 13, Issue 4, April 2024

**ISSN** INTERNATIONAL  
STANDARD  
SERIAL  
NUMBER  
INDIA

Impact Factor: 8.423

 9940 572 462

 6381 907 438

 [ijirset@gmail.com](mailto:ijirset@gmail.com)

 [www.ijirset.com](http://www.ijirset.com)

# Natural Language to Code: Enhancing Source Code Generation with GPT Integration

Dr. Madhuri Kovoor<sup>1</sup>, Shaista Firdous<sup>2</sup>, M. Sathvika<sup>3</sup>, K. Shiva Sai<sup>4</sup>

<sup>1</sup>: Associate Professor, Dept. of CSE, Anurag University, Hyderabad, India

<sup>2</sup>: UG Student, Dept. of CSE, Anurag University, Hyderabad, India

<sup>3</sup>: UG Student, Dept. of CSE, Anurag University, Hyderabad, India

<sup>4</sup>: UG Student, Dept. of CSE, Anurag University, Hyderabad, India

**ABSTRACT:** In today's evolving technology environment, the need for a good software development process has never been greater. The traditional way of using code coding software often encounters problems in meeting the speed and work speed needs of today's projects. To solve these problems, there is growing interest in using artificial intelligence (AI) and natural language processing (NLP) technologies to automate some coding processes. The project delves deep into the world of AI-driven code generation and uses the powerful GPT-3 Transformer model as its cornerstone. The fine-grained model uses proprietary data to understand and generate code snippets based on language clues. This new approach has the potential to increase software development performance, providing developers with the tools to quickly prototype, debug, and develop code with greater efficiency and accuracy. With this guidance, we begin the journey of finding the power of artificial intelligence to improve the creative process of business and ultimately create the future of software engineering.

**KEYWORDS:** GPT-3, transformer model, code generation, fine-tuning, natural language processing, machine learning.

## I. INTRODUCTION

Exploring the realms of intelligence development, technology plays a pivotal role in generating code. In the fast-paced world of software development, the need for streamlined and efficient coding practices has never been higher. Traditional methods often involve manual coding, which can be time-consuming and error-prone. To solve these problems, the integration of artificial intelligence (AI) has emerged as a good way to increase the diversity of the coding process. At the forefront of this intersection is the GPT-3 Transformer model, known for its ability to understand and create human-like texts. However, its application in code generation must be fine-tuned based on specific data to ensure accuracy and accuracy. The project aims to delve deeper into this complex combination of intelligence and programming, taking into account the full capabilities of GPT-3 to automatically generate code snippets. By training the model on selected data specific to the programming domain, we seek to improve its understanding of coding standards, conventions, and best practices. Through a combination of reports and machine learning algorithms, our goal is to give developers tools that can turn their ideas into actionable code. This introduction paves the way for extensive research on the methods, problems, and consequences of using GPT-3 to generate code, ultimately shaping the future of software development.

## II. RELATED WORK

**Ahmed Sadik et al[1]** demonstrated the potential of large language models (LLMs) such as ChatGPT to help improve software performance. Author LL.M. It can make people useful, but it cannot replace their skills. This article explores the field of LL.M. It can perform different tasks such as code generation, data generation, and error detection. They demonstrate the potential of the LL.M. Update software development by addressing advances in areas that show deep learning's past limitations have been overcome. **Man Fai Wong et al[2]** investigated how large language models (LLMs) learned in large numbers (large numbers) can be replaced by intelligent assisted programming. Using the idea that numbers are similar to natural languages, LLMs can perform tasks such as generating numbers, executing them, and looking for errors. This article explores how this differs from previous LLM studies and includes a review of the extensive literature, evaluations, and recent developments in AI-assisted programming for Transformer-based LLMs. He also talked about future challenges and opportunities. **Jiho Shin et al [3]** is working on a new programming method: using non-traditional programming languages. This method must overcome limitations such as high tuition fees and

limited teaching capacity. This article reviews existing research on automatic code generation from annotations. It divides this process into input and output, analyzes current trends and reports future trends. The authors recommend focusing on language modeling rules and searching for better representatives of programming techniques such as embedding techniques and pre-learning models (implemented in NLP).

**Tapornoy Adhikari et al [4]** studied how natural language processing (NLP) can be used to identify program code based on description. It highlights benefits such as less effort and increased productivity, but also acknowledges challenges such as ambiguous language and complex numbers. This article reviews current methods, evaluations, and future directions, including exploring advanced machine learning techniques and combining NLP with other methods. Overall, research shows that NLP has the potential to revolutionize software development. **Hendrig Sellik [5]** research papers on how to use natural language processing (NLP) to generate code programs from descriptions. NLP techniques such as word embedding and deep learning models are used for tasks such as generating code comments, executing code written by developers, and even generating names for jobs and classes. Although many generations still face problems, NLP holds promise in developing software for better results. **Sethunya R Joseph et.al [6]** Research papers on natural language processing (NLP), a branch of computer science dedicated to the computer's understanding and manipulation of human speech. NLP draws on disciplines as diverse as linguistics and computer science to perform tasks such as machine translation and transcription. This article explores the history of the science of NLP, its increasing importance due to the rise of the Internet and the need for software that can process text in many languages.

### III. PROPOSED METHOD

The proposed methodology has demonstrated a significant enhancement in ROUGE score indicating that the source code generated efficiently points towards the user query and fulfills the requirements of the user in a most accurate manner. Additionally, it has overcome the problem of language barrier and the understandability of the user towards the source code.

**Fine-tuning the model:** Fine-tuning enhances the capabilities of models accessible through the API by delivering superior results compared to prompting, accommodating training on a larger volume of examples beyond prompt constraints, achieving token savings through concise prompts, and enabling requests with lower latency. The fine-tuning process encompasses the following stages:

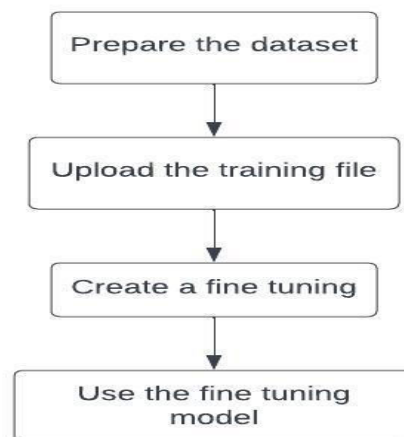
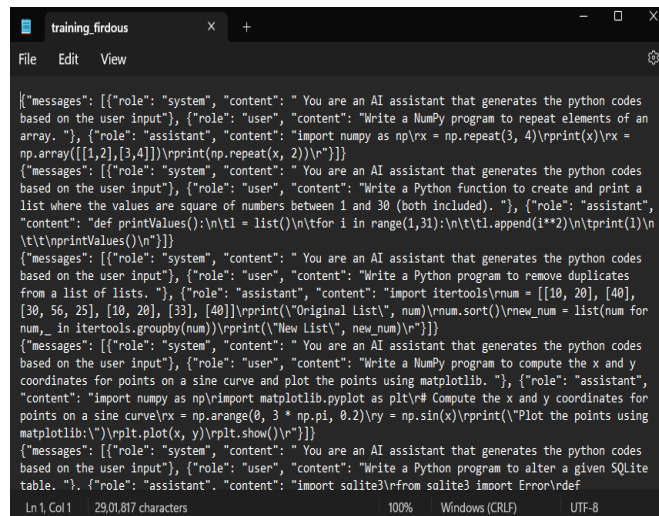


Figure 1: Stages of Fine-tuning a model

**Prepare and upload training data:** It is necessary to organize the dataset in a conversational format, similar to the structure used by the Chat Completions API. This involves presenting the data as a series of messages, where each message specifies a role and contains the actual message content. To achieve this, the dataset needs to be converted into a JSON Lines file format as shown in figure 3.1.1.



```

{"messages": [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a NumPy program to repeat elements of an array. "}, {"role": "assistant", "content": "import numpy as np\nrx = np.repeat(3, 4)\nprint(x)\nrx = np.array([[1,2],[3,4]])\nprint(np.repeat(x, 2))\n"}]}
{"messages": [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a Python function to create and print a list where the values are square of numbers between 1 and 30 (both included). "}, {"role": "assistant", "content": "def printValues():\n\tl = list()\n\tfor i in range(1,31):\n\t\tl.append(i**2)\n\tprint(l)\n\t\t\n\tprintValues()\n"}]}
{"messages": [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a Python program to remove duplicates from a list of lists. "}, {"role": "assistant", "content": "import itertools\nrnum = [[10, 20], [40], [30, 56, 25], [10, 20], [33], [40]]\nprint("\nOriginal List", rnum)\nrnum.sort()\nrnew_num = list(num for num, _ in itertools.groupby(rnum))\nprint("\nNew List", new_num)\n"}]}
{"messages": [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a NumPy program to compute the x and y coordinates for points on a sine curve and plot the points using matplotlib. "}, {"role": "assistant", "content": "import numpy as np\nimport matplotlib.pyplot as plt\n# Compute the x and y coordinates for points on a sine curve\nrx = np.arange(0, 3 * np.pi, 0.2)\nry = np.sin(rx)\nprint("\nPlot the points using matplotlib:\n")\nplt.plot(x, y)\nplt.show()\n"}]}
{"messages": [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a Python program to alter a given SQLAlchemy table. "}, {"role": "assistant", "content": "import sqlalchemy\nfrom sqlalchemy import Engine\n"}]}

```

Figure 2: Training Data

After preparing the training data, the next step is to upload it using the Files API. This allows the data to be utilized in fine-tuning jobs effectively. Once everything is done we can start the finetuning of the model.

**Create a Fine Tune Model:** After uploading the file, the subsequent action is to initiate a fine-tuning job. This is accomplished by creating a fine-tuning job through the OpenAI SDK:

**Step-1:** Create an instance of the OpenAI client for interactions.

**Step-2:** Utilize the OpenAI client to create a fine-tuning job.

**Step-3:** Specify the training file ID that was returned when the training file was uploaded to the OpenAI API as its identifier (e.g., "file-abc123")

**Step-4:** Indicate the model to be used for fine-tuning as "gpt-3.5-turbo."

Upon commencing a fine-tuning job, the time for completion may fluctuate based on queue dynamics and the duration of the training process, which is contingent on factors like the model and dataset size. After the training concludes, users will be notified with a confirmation mail.

Once the job has been successfully completed, you'll observe that the fine-tuned model field is filled with the model's name when retrieving the job details.

The overall algorithm for fine tuning:

**Step 1:** Import pandas

**Step 2:** Upload the csv dataset

**Step 3:** Encode the dataset with 'cp1252'

**Step 4:** Create a new jsonl file named 'training.jsonl' and open it in writing mode

**Step 5:** Create a function CREATE\_DATASET which takes a judgment and its corresponding summary as the input parameters.

**Step 6:** The function creates a message as shown in the above example and returns it.

**Step 7:** Iterate through the CSV file and send the judgment and summary one by one to CREATE\_DATASET.

**Step 8:** While iterating write the returned message (every time in a new line) from CREATE\_DATASET in training.jsonl.

**Step 9:** From 'openai' library import 'OpenAI' class

**Step 10:** Create an instance of the OpenAI class as 'client' with an 'api\_key' parameter set as user API KEY, which will be used to communicate with the OpenAI API.

**Step 11:** Upload the training file using 'file.create()' method under 'OpenAI' and set its 'file' parameter as open 'training.jsonl' file in read binary mode and set the 'purpose' parameter as 'fine-tune'. A file id will be generated

**Step 12:** Fine tune the model using 'fine\_tuning.jobs.create()' method under OpenAI class and set its parameters. Set 'training\_file' as the file id generated after uploading the training file, set the 'model' as the 'gpt-3.5-turbo'. This will create a fine-tuning job id.

**Step 13:** Retrieve the fine-tuned model status using fine\_tuning.job.retrieve() method under OpenAI. The model name will be generated.

**Generating source codes using fine-tuned model:** Once the finetuning job is done our finetuned model is ready to be used in out to generate the source codes for the python language. Our system leverages a fine-tuned GPT-3 model to

assist in Python code generation. This means we've taken the powerful GPT-3 language model and further specialized it to understand the syntax and structure of Python code. By providing the model with clear prompts and descriptions of the desired functionality, we can generate Python code snippets that automate tasks or implement specific algorithms. This in-house development tool accelerates our workflow and frees up valuable resources for our team.

The usage of the finetuned model goes something like this:

```
question = st.text_input("Type your question here...")
response = client.chat.completions.create(
    model= <Finetuned model name>,
    messages = [{"role": "user", "content": question}],
)
Client = OpenAI(api_key=YOUR_OPENAI_API_KEY)
generated_code = response.choices[0].message.content
```

Here 'question' is the variable which holds the users query  $Q_a$  that is the question and when the user gives the input to the system the response function comes to the action where its takes the  $Q_a$  as the input for itself and generates a code  $G_c$  for it once the response is generated it is passed on to the function 'generate\_algorithm()' where the algorithm for the  $G_c$  is stored in *Generated\_algorithm*  $G_a$ .

### Creating a language translation model:

#### Dataset Acquisition and Preprocessing:

**Dataset:** We utilize the "cfilt/iitb-english-hindi" dataset from the Hugging Face Datasets library. This dataset provides parallel English and Hindi sentences for training the translation model.

**Preprocessing:** A preprocessing function is defined to handle the following:

- Extract English and Hindi sentences from the dataset.
- Tokenize the sentences using a pre-trained tokenizer from the Helsinki-NLP/opus-mt-en-hi model checkpoint. This tokenizer converts text into numerical representations suitable for the machine translation model.
- Set a maximum length for both English and Hindi sentences to ensure efficient model training.

#### Model Architecture and Training:

**Pre-trained Model:** The core of our translation model is the TFAutoModelForSeq2SeqLM class from Transformers. This class provides a pre-trained sequence-to-sequence model architecture specifically designed for machine translation tasks. The chosen pre-trained model checkpoint, "Helsinki-NLP/opus-mt-en-hi", is pre-trained on a massive dataset of English-Hindi translations.

#### Training Configuration:

- Batch Size:** This defines the number of sentences processed together during training. We use a batch size of 16.
- Learning Rate:** This hyperparameter controls the speed of learning for the model. We use a learning rate of  $2e-5$ .
- Weight Decay:** This helps prevent overfitting by reducing the influence of large weights during training. A weight decay of 0.01 is used.
- Epochs:** We train the model for 15 epochs, where one epoch represents a complete pass through the entire training dataset.

**Training Loop:** The model.fit function is used to train the model on the prepared training dataset. During training, the model iterates through batches of data, learns to translate English sentences to Hindi, and updates its internal parameters to improve translation accuracy.

#### Inference and Model Saving:

- Saving the Model:** After training, the fine-tuned model is saved for later use. This allows us to load the trained model for translation tasks without re-training.
- Translation Function:** The model.generate function is used to translate new English sentences. The input text is tokenized and fed into the model, which generates the corresponding Hindi translation.
- Tokenizer Configuration:** During translation, the tokenizer is used again to convert the model's output (numerical representation) back into human-readable Hindi text.

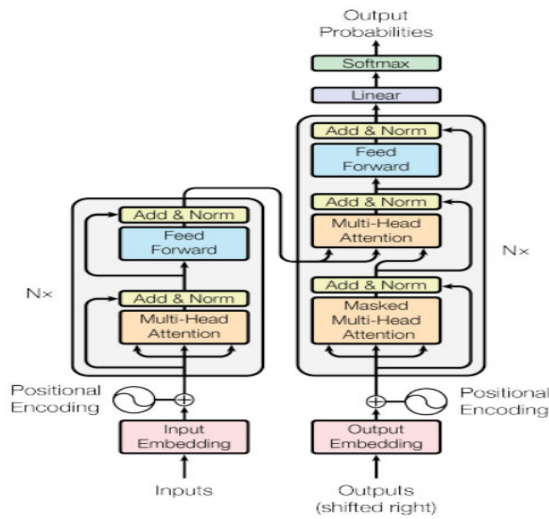


Figure 3: Language Translation Model

The figure 3.3 showcases a simplified view of a Transformer model in action for machine translation. Imagine an English sentence entering on the left. The encoder, like a detective, dissects the sentence, analyzing each word's meaning and how it connects to others. This analysis is captured by those blue boxes with swirling arrows. Meanwhile, on the right, the decoder acts like a writer crafting the Hindi translation. It starts with a special token (like a "start of sentence" marker) and, using its own internal attention (represented by orange boxes with curvy arrows), builds the translation. But here's the clever part: the decoder also peeks back at the encoder's analysis (through the double-headed arrows) to ensure each translated word aligns with the original meaning in English. This interplay between analyzing the source language and attentively generating the target language is what makes Transformers powerful for machine translation tasks.

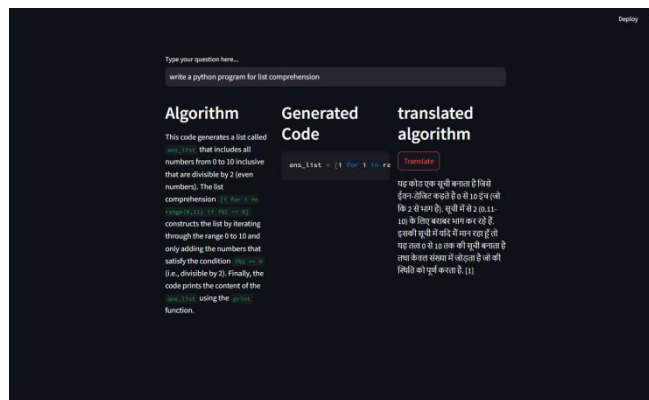


Figure 4: User Interface

The above image shows the user interface where there is a text input section for the user to enter his question or the query. According to the query the output that is the python code along with the algorithm is generated. There is a translate button given for the convenience of the user to translate the explanation of the code into Hindi language.

#### IV. EXPERIMENT RESULTS

For assessing the effectiveness of system-generated codes, we employed the ROUGE metric, an acronym for Recall-Oriented Understudy for Gisting Evaluation. In this context, "Gisting" refers to the extraction of the primary point from the text [31]. ROUGE serves as an evaluation matrix that compares the generated code with a reference code. A higher ROUGE score signifies a superior alignment between the code and the reference code. ROUGE comprises five variants, including ROUGE-N, ROUGE-S, ROUGE-L, ROUGE-W, and ROUGE-SU [32]. In this study, we specifically employed ROUGE score metrics to analyse various code generation models, where N denotes the length of the ngram,



encompassing ROUGE-1 (unigram), ROUGE-2 (bigram), ROUGE-3 (trigram), and so forth. The definition of ROUGE-N or ROUGE-L Metrics in terms of precision, recall, and F1 Scores parameters is elucidated as follows.

Precision in code generation refers to the accuracy of generated code snippets compared to a reference set of correct code.

$$\text{Precision} = \frac{\text{Number of overlapping tokens between generated and reference code}}{\text{Total number of tokens in generated code}}$$

Recall measures the proportion of relevant or correct tokens present in the reference code that are successfully captured by the generated code.

$$\text{Recall} = \frac{\text{Number of overlapping tokens between generated and reference code}}{\text{Total number of tokens in reference code}}$$

The F1 score represents the harmonic mean of precision and recall, amalgamating both measures into a single score that achieves a balance between precision and recall, as demonstrated in Equation.

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Table I presents the average precision scores of various code generation models tested on CoNaLa dataset. Our observation indicate that the proposed method excels, achieving a precision score of 0.7 surpassing other methods.

TABLE I: AVERAGE PRECISION SCORES OF DIFFERENT MODELS

Models	Precision
MarianCG	0.57
RoBERTaMarian	0.40
BERTMarian	0.47
ELECTRAMarian	0.34
LUKEMarian	0.53
Proposed	0.7

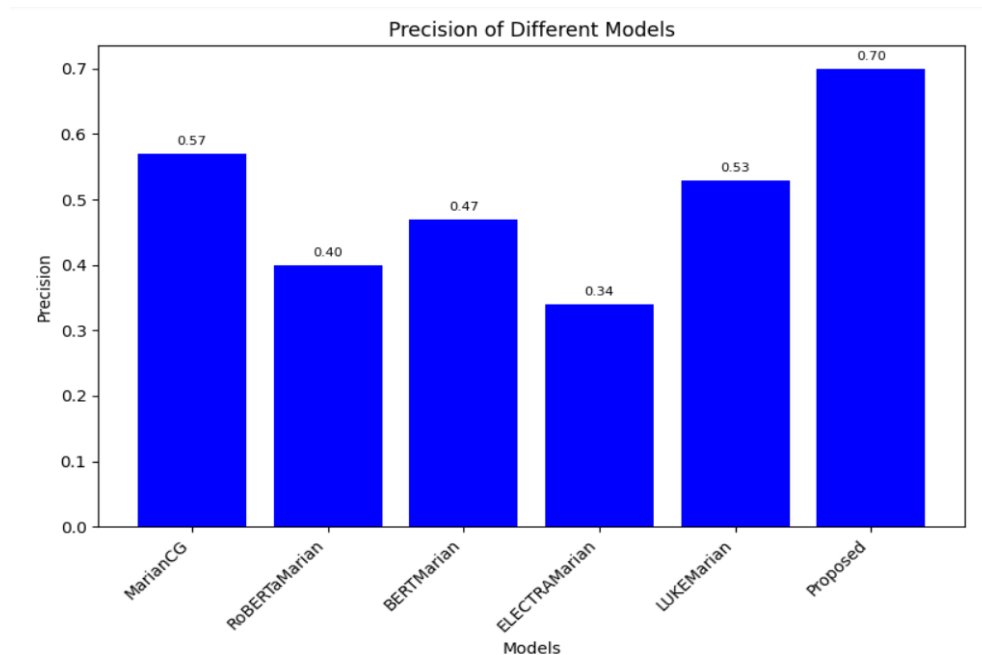


Figure 5: Precision Score Comparison



Table II displays the Accuracy score for various code generation models. When we examine the table it is evident that the proposed model has the highest accuracy stating that the code from the generated codes match nearly with the reference codes.

TABLE II: ACCURACY SCORES OF DIFFERENT MODELS

S.No.	Models	Accuracy
1.	MarianCG	10.2
2.	RoBERTaMarian	13.8
3.	BERTMarian	12.4
4.	ELECTRAMarian	10.0
5.	LUKEMarian	7.6
6.	Proposed	15.23

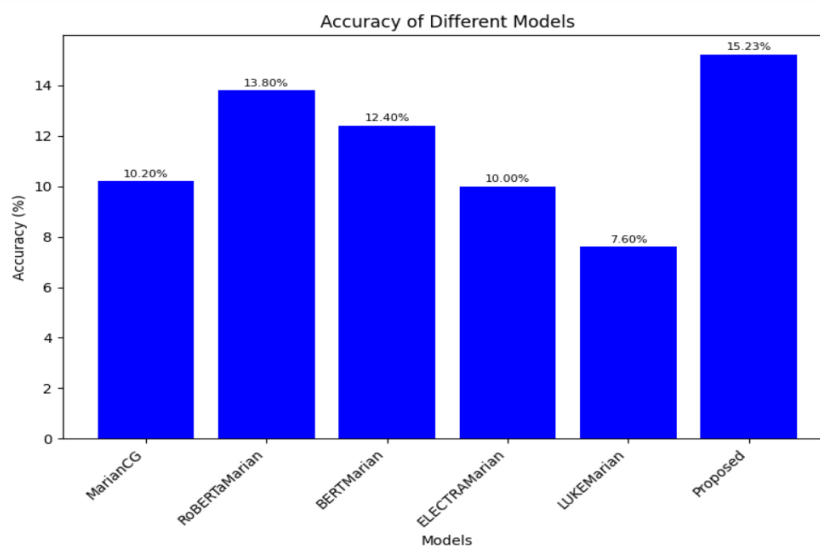


Figure 6: Accuracy Score Comparison

Table III displays the Average Rouge scores of code generation models on CoNaLa dataset. According to the table, the proposed method achieves the highest ROUGE scores.

TABLE III: ROUGE SCORES OF DIFFERENT MODELS

S.No.	Models	Rouge Score
1.	MarianCG	49.62
2.	RoBERTaMarian	44.25
3.	BERTMarian	43.94
4.	ELECTRAMarian	42.42
5.	LUKEMarian	39.84
6.	Proposed	51.23



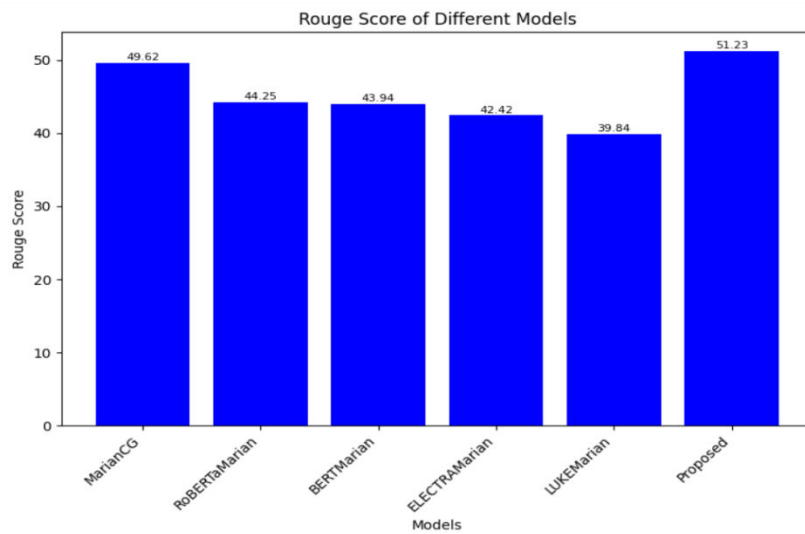


Figure 7: Rouge Score Comparison

TABLE IV: BLUE SCORES OF DIFFERENT MODELS

S.No.	Models	Rouge Score
1.	MarianCG	34.42
2.	RoBERTaMarian	35.73
3.	BERTMarian	32.46
4.	ELECTRAMarian	30.18
5.	LUKEMarian	29.82
6.	Proposed	37.23

Table IV displays the BLUE scores of code generation models on CoNaLa dataset. According to the table, the proposed method achieves the highest BLUE scores.

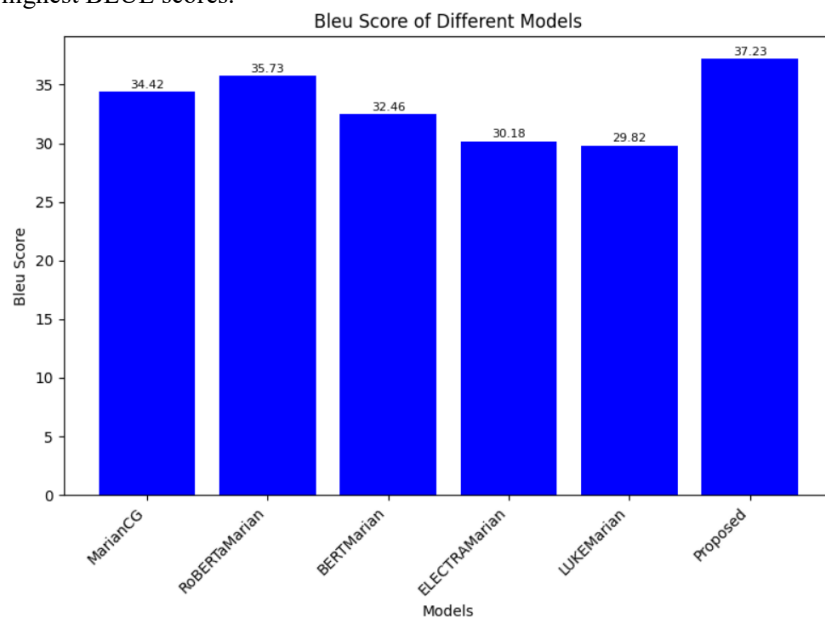


Figure 8: BLEU Score Comparison

When see the above results clearly tell that proposed method excels in its task when compared to the previous methods.

## V. CONCLUSIONS

The research has proven that code generation of the proposed model compared to the already existing models has been more accurate and efficient. Unlike the previous models having the limitations in generating code and having the legitimate response guarantee at stack proposed method overcomes these by only restricting its knowledge base to a specific dataset. Through the finetuning of GPT-3 transformer with the large dataset makes the code generation model excel in its performance.

## REFERENCES

- [1] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy, Pages 754-768. IEEE, 2022.
- [2] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In International Conference on Learning Representations, 2022
- [3] Singh, S.M., Meetei, I.S., Singh, T.D., Bandyopadhyay, S., 2021b. Multiple captions. embellished multilingual multi-modal neural machine translation. In: Proceedings of the First Workshop on Multimodal Machine Translation for Low Resource Languages. MMTLRL 2021, pp. 2-11, <http://aclanthology.org/2021.mmturi-1.2>.
- [4] Ramesh, G., Doddapaneni, S., Bheemaraj, A., Jobanputra, M., AK, R., Sharma, A Sahoo, S., Diddee, H., Kakwani, D., Kumar, N., et al., 2022. Samanantar: The largest publicly available parallel corpora collection for 11 indic languages. Trans. Assoc Comput. Linguist, 10, 145-162 <http://dx.doi.org/10.1162/tacil.a.00452>.
- [5] Elazar Y, Kassner N, Ravfogel S, Ravichander A, Hovy E, Schütze H, Goldberg Y (2021) Erratum: measuring and improving consistency in pretrained language models. Trans Assoc Comput Linguist9:1407. <https://doi.org/10.1162/tacl.x.00455>
- [6] Raffel C, Shazeer NM, Roberts A, Lee K, Narang S, Matena M, Zhou Y, LiW, Liu PJ (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. ArXiv arXiv:1910.10683
- [7] Gad W, Alokla A, Nazih W, Salem AB, Aref M (2021) DlbDeep learning-based transformer to generate pseudo-code from source code. CMC 70:3117–3123. <https://doi.org/10.32604/cmc.2022.019884>
- [8] Alokla A, Gad W, Nazih W, Aref M, Salem AB (2022) Retrievalbased transformer pseudocode generation. Mathematics 10(4):604. <https://doi.org/10.3390/math10040604>
- [9] Kaur P, Kumar H, Kaushal S (2023) Technology-assisted language learning adaptive systems: a comprehensive review. Int J Cogn Comput Eng 4:301-313 <https://doi.org/10.1016/j.ijcce.2023.09.002>
- [10] .Javidpanah M, Javadpour A, Rezaei S (2021) ROOA: CloudIDE framework for extension development. Int J Cogn Comput Eng 2:165–170. <https://doi.org/10.1016/j.ijcce.2021.09.003>
- [11] Guizzo G, Zhang J, Sarro F, Treude C, Harman M (2023) Mutation analysis for evaluating code translation. Empir Softw Eng 29:19
- [12] Athiwaratkun B, Gouda SK, Wang Z, Li X, Tian Y, Tan M, Ahmad WU, Wang S, Sun Q, Shang M, Gonugondla SK, Ding H, Kumar V, Fulton N, Farahani A, Jain S, Giaquinto R, Qian H, Ramanathan MK, Nallapati R, Ray B, Bhatia P, Sengupta S, Roth D, Xiang B (2023) Multi-lingual evaluation of code generation models. arXiv preprint arXiv:2210.14868
- [13] Dahal S, Maharana A, Bansal M (2021) Analysis of tree-structured architectures for code generation. In: Findings of the association for computational linguistics: ACL-IJCNLP 2021, pp 4382–4391
- [14] Qin P, Tan W, Guo J, Shen B, Tang Q (2021) Achieving semantic consistency for multilingual sentence representation using an explainable machine natural language parser (mparser). Appl Sci 24:11699
- [15] Tang Z, Shen X, Li C, Ge J, Huang L, Zhu Z, Luo B (2022) Asttrans: code summarization with efficient tree-structured attention. In: 2022 IEEE/ACM 44th international conference on software engineering (ICSE), pp 150–162
- [16] Shin R, Lin CH, Thomson S, Chen C, Roy S, Platanios EA, Pauls A, Klein D, Eisner J, Van Durme B (2021) Constrained language models yield few-shot semantic parsers. arXiv preprint arXiv:2104.08768
- [17] Lano K, Xue Q (2023) Code generation by example using symbolic machine learning. SN Comput Sci 4:1–23
- [18] Le THM, Chen H, Babar MA (2020) Deep learning for source code modeling and generation. ACM Comput Surv (CSUR) 53:1–38
- [19] Norouzi S, Tang K, Cao Y (2021) Code generation from natural language with less prior knowledge and more monolingual data. In: Proceedings of the 59th Annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 2: short papers), pp 776–785.
- [20] Orlanski G, Gittens A (2021) Reading stackoverflow encourages cheating: adding question text improves extractive code generation. arXiv preprint arXiv:2106.04447



INTERNATIONAL  
STANDARD  
SERIAL  
NUMBER  
INDIA



# INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

 9940 572 462  6381 907 438  [ijirset@gmail.com](mailto:ijirset@gmail.com)



[www.ijirset.com](http://www.ijirset.com)

Scan to save the contact details