STM JOURNALS

Recent Trends in
Programming Languages

ISSN: 2455-1821
Volume 12, Issue 3, 2025
September–December
DOI (Journal): 10.37591/RTPL

https://journals.stmjournals.com/rtpl

Review RTPL

Encoder-Decoder Based Fine-Tuned Model for Code Doubt Solver

Utkarsh Yadav*, Shivesh, Shivam Shukla, Ujjwal Sharma, Richa Suryavanshi

Abstract

As we are growing in technology, more technologically skilled persons are needed in industry. They all often rely on programming in their daily work, and when some doubts arise, they seek help from teachers to LLMs like GPT to Deepseek. However, when errors arise, then comes hectic part to troubleshoot and resolve the error. Usually, people seek help from some LLMs like GPT, or Deepseek for the solution; they give the solution, but they are general purpose model, which means they are not optimized for coding purpose, so they may lack in accuracy. Although most current models, rely on an encoder-only or decoder-only, and also the training of model is suboptimal for generation and code interpret tasks and process code snippets similar to natural language, ignoring programming language-specific symbols during tokenization. These all pre-tuned models are transformer based, either of Encoder based or Decoder based (e.g. BERT, GTP, RoBERTa, LLaMa, etc.) which make them capable of either understanding or generation individually. As Encoder based model is effective for understanding tasks only, not ideal for generation purpose, while Decoder based model is effective in generation from prompt. And this makes the pre-trained model inefficient for code debugging, code explanation, code completion and code generation purpose. So ideally, we should use Encoder-Decoder based transformer model (e.g. mT5, BART, T5, etc.) as the encoder comprehends the input, and the decoder generates relevant output, which makes it best for Code generation, Code completion, Code explanation and Code debugging. This study explores the optimization of encoder-decoder transformer architectures for code-related tasks by fine-tuning them specifically on programming language datasets. We evaluate their effectiveness in understanding and generating syntactically correct and semantically meaningful code, as well as solving user-generated coding doubts, code completion, code generation and code debugging. Our proposed method not only improves effectiveness in understanding but it also improves accuracy and optimizes outcome for coding purpose such as general programming doubt, code debugging, code generation, code completion, code explanation and much more. Experimental results demonstrate that the fine-tuned encoder-decoder model is outperforming the general purpose LLM, as it is better and more capable in Code correction, Code completion, Code generation, Code explanation, etc. The Encoder-Decoder model which is perfectly fine-tuned with dataset, that model can outperform and intelligently respond in the coding purpose, offering more reliable and accurate response. This research work lays the foundation to bridge the gap between Natural Language (NL) and Programming Language (PL).

***Author for Correspondence**
Utkarsh Yadav
E-mail: it.is.utkarsh.here@gmail.com

Student, Department of Computer Science and Engineering, Echelon Institute of Technology, Faridabad, Haryana, India

Received Date: July 09, 2025
Accepted Date: October 07, 2025
Published Date: October 17, 2025

Citation: Utkarsh Yadav, Shivesh, Shivam Shukla, Ujjwal Sharma, Richa Suryavanshi. Encoder-Decoder Based Fine-Tuned Model for Code Doubt Solver. Recent Trends in Programming Languages. 2025; 12(3): 35–42p.

Keywords: Encoder-decoder transformer, code generation, program repair, fine-tuning, natural language to code (NL2Code)

INTRODUCTION

Programming assistant has become a crucial need nowadays, as we suffer in any programming related doubt. We need some Programming assistant to help us. Well, there are many models that can act as programming assistants like GPT; they have shown remarkable general capability, but somewhere they

like accuracy in programming purposes like code generation, code completion and understanding coding specific character and syntax. So, we aim to fix this gap.

Limitations of the Current Models

a. Most models are general purpose models, which means they are not fully optimized for coding or programming purpose. They are best for general purpose but they lack in coding purpose.
b. Normally, models work on Natural Language (NL), during tokenization, they usually ignore programming syntax, resulting into reduced accuracy of the generated code.
c. Encoder-only model or Decoder-only model, mostly models are either Encoder only or Decoder only as they are best in understanding and generation respectively.

Need for fine-tuning on Models

a. Fine-tuning of the model can significantly enhance the ability to generate correct, efficient, accurate and relevant code.
b. Encoder-Decoder model is already best for understanding and generation purpose, fine-tuning it makes it suitable for programming assistant.

Role of Encoder-Decoder Models

a. Encoder-only and Decoder-only model is ideal for understanding and generation respectively.
b. Encoder-Decoder models like T5, BART etc. have bidirectional encoding mechanism, and they are capable of deeper semantic understanding and are highly accurate and relevant for generation.

Research Objectives

a. This research focuses on increased effectiveness of the model for programming purpose by fine-tuning the encoder-decoder model, to perform programming related tasks effectively.
b. It aims to increase productivity than most of the general purpose models, which work on Natural Language (NL) and optimize it fully for the Programming Language (PL), which makes it efficient and reliable for coding and programming purpose.

LITERATURE SURVEY

Well, there are many models which can debug and generate code, like GPT, DeepSeek, BERT, RoBERTa and many others. In 2020, Feng *et al*. researched on Natural Language Model and Programming Language [1].

According to that, currently most models are general purpose models trained for Natural Language (NL). Although these models get the work done as they can solve most of the doubts and generate code and debug code snippet, but problem with these models is that as these are trained for Natural Language (NL), they ignore the special characters of Programming Language (PL) during tokenization, which reduces the accuracy of the generated and debugged code. This is why these are sub-optimal for code generation and debugging purpose.

In 2020, Wang *et al*. did research on these models based on transformer architecture [2]. The transformer gave them self-attention mechanism. These transformer architecture based models can be classified into three categories:

i. Encoder model,
ii. Decoder model, and
iii. Encoder-Decoder model.

Encoder-only Models

These models, such as BERT, RoBERTa, and ELECTRA, encode input sequences into vector representations. They are effective for understanding tasks like classification but are not ideal for generating content.

Decoder-only Models

These models, including GPT, LLaMA, and CodeGen, are designed to generate sequences from scratch. However, they often lack strong input comprehension capabilities, making them suboptimal for tasks that require deep understanding of the input context.

Encoder-Decoder Models

These models combine the strengths of both: the encoder comprehends the input, and the decoder generates relevant output. Examples include T5, CodeT5, BART, mT5, and MASS. They are especially effective for tasks such as summarizing code, translating between programming languages, and identifying or fixing bugs.

Moreover, the majority of current approaches apply standard NLP pre-training methods to source code, treating it as a sequence of tokens similar to natural language rather than acknowledging its unique structure as a programming language. This largely ignores the rich structural information in code, which is vital to fully comprehend the code semantics. This significantly limits their performance in real-world programming scenarios [3].

In 2021, Chen *et al.*, [4] researched on same topic, they investigated GPT-3 it could only generate simple program from Python docstring.

They then fine-tuned a version of GPT called Codex. The performance of their models on the HumanEval dataset, measured by pass rates, varies with model size. When generating a single solution per problem, GPT-12B fails to solve any problems. In contrast, Codex, fine-tuned specifically on code, achieves a 28.8% success rate, while Codex-S, which undergoes additional fine-tuning on correctly implemented standalone functions, reaches a 37.7% pass rate. From here, further gains can be realized by generating 100 samples per problem and selecting the sample with the highest mean log-probability (44.5% solved) or by selecting the sample that passes the unit tests (77.5% solved). All samples are generated with temperature. So, from here we clearly see before fine-tuned and after fine-tuned model accuracy results, as shown in Figure 1. So, to generate more accurate and efficient results, we should optimize the model for coding purpose, and best way is to fine-tune the model.

In 2021 Austin *et al.* evaluated a general purpose model for coding purpose [5]. The models were evaluated on two benchmarks: MBPP and MathQA-Python. These benchmarks are designed to assess a model's ability to generate short Python programs from natural language descriptions.

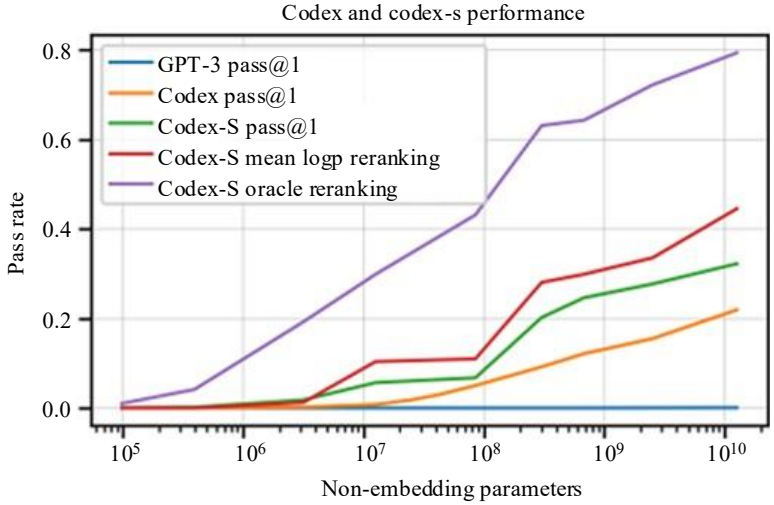

Figure 1. Comparison of code generation performance across models with varying parameters.

The Mostly Basic Programming Problems (MBPP) dataset consists of 974 tasks intended for entry-level programmers. In contrast, the MathQA-Python dataset, a Python adaptation of the MathQA benchmark, includes 23,914 problems that test the model's capability to generate code from more complex and structured textual inputs.

Their largest models, even without being fine-tuned on a coding dataset, are able to solve 59.6% of the problems in the MBPP benchmark using few-shot learning with an effective prompt. Fine-tuning on the dataset boosts performance by approximately 10 percentage points across nearly all model sizes. On the MathQA-Python dataset, the largest fine-tuned model reaches an accuracy of 83.8%. Additionally, they investigate the models' semantic understanding by fine-tuning them to predict the outcomes of program execution.

In 2022 Nijkamp *et al*. evaluated the performance of the model [6]. They trained the model on large dataset on Natural Language and Programming Language and also used multi-turn program synthesis approach. The study shows that using multi-turn program approach significantly improves program synthesis resulting more precise solution.

In 2021 Guo *et al.* researched on the existing model, in regards to a code snippet as a sequence of tokens, while ignoring the inherent structure, which gave code semantics and would improve the code understanding [7]. They incorporated data flow during the pre-training stage, a semantic-level representation of code that captures the relationships between code elements and syntax. This approach enhances the model's understanding of code structure and behavior during processing.

In 2023, Piskounova [8] and Zhang *et al.* [9] researched on learning based automatic program repair. Automatic program repair (APR) aims to fix the bugs automatically. With advancement in deep learning (DL), APR has been proposed to neural network to learn bug-fixing pattern. APR finds the buggy code and fixes the code snippets from learning technique from Neural network.

In 2021, Lu *et al*. introduced CodeXGLUE, a benchmark dataset to foster machine learning research for program understanding and generation [10]. CodeXGLUE includes platform for model evaluation and comparison. It also features three styles: BERT-style, GTP-style, and Encoder-Decoder Style. Their model CodeXGLUE, supports the following tasks:

1. *Code-Code:* This include Code completion, Code debugging, Code repair and more.
2. *Text-Code:* This make it able to search for code from natural language.
3. *Code-Text:* This can be used for Code explanation.
4. *Text-Text:* This can be used to explain the documentation.

In 2021 Shah [11] and in 2023 Li *et al.* [12] studied and researched to automatically fixing compilation error and how it impacts productivity of development. This proposes end to end solution, Transrepair to locate the error lines and create the correct substitute for the program. A transformer-based neural network is used to learn how to repair code by leveraging the erroneous code itself, its surrounding context, and the diagnostic feedback. This enables the model to generate effective fixes by understanding both the error and its broader programming environment.

PROPOSED METHODOLOGY

To overcome the limitations of general-purpose NLP models which is sub-optimal in code generation and code debugging related tasks, we propose a fine-tuned encoder-decoder transformer model optimized specifically for code understanding, generation, and debugging [13–18]. Our methodology involves the following steps:

Model Selection

We select a pre-trained encoder-decoder model, which makes it able to easily understand prompt through encoder and generate response through decoder such as CodeT5 or T5 due to their balanced understanding and generation capabilities.

Dataset Collection

We gather a curated dataset consisting of:

- Code snippets with bugs and corresponding fixed versions;
- General programming doubts and answers;
- Code explanations and comment generations;
- Code and their complexities; and
- Code and their optimized version.

Data Preprocessing

- Tokenize code using a tokenizer that preserves special characters (like angle brackets, colons, etc.).
- Normalize code structure if necessary (indentation, syntax correction).
- Align input-output pairs form the dataset for training (e.g., buggy → fixed, question → answer).

Fine-Tuning

- We fine-tune the base model on the curated dataset using supervised learning.
- Use loss functions such as cross-entropy to optimize accuracy in generation.

Evaluation Metrics

- Use BLEU, ROUGE, and Exact Match Score to evaluate generation quality.
- Use code execution accuracy (pass/fail) for debugging tasks.

Deployment

The model can be wrapped in a simple interface or chatbot for interactive use by learners and developers.

EXPECTED OUTCOME

The expected outcomes of our approach are as follows:

Enhanced Accuracy in Code Generation and Debugging

By fine-tuning an encoder-decoder transformer model specifically for code generation and code debugging, we expect better handling of programming syntax, special characters, and structure leading to more accurate, improved and syntactically correct code generation and debugging [19–22].

Improved Programming Language Understanding

The model will be capable of understanding user queries related to code and providing meaningful and contextual explanations, helping users resolve their programming doubts efficiently.

Enhanced Model Performance Compared to General NLP Models

Our model is expected to outperform general-purpose NLP models like GPT or BERT in code-related tasks, particularly in:

- Bug fixing,
- Function completion,
- Explaining code logic,
- Code generation, and
- Optimizing code.

As this uses an encoder-decoder transformer model, it makes them more capable of understanding queries and generating accurate and improved responses compared to regular NLP models [23–25].

Reduction in Developer Time and Effort

With an AI assistant capable of automatically understanding and resolving code issues, developers can reduce debugging time and focus more on logic building and application-level tasks, instead of spending time for debugging the code.

Real-World Applicability

The solution can be integrated into development environments or online platforms to assist students, educators, and software developers with real-time code support. Which they can use to solve their query, from getting answer of general coding doubts to code debugging.

Multitasking Capability

As this model is encoder-decoder based this can easily switch between encoder model to decoder model, making it seamlessly switching between tasks like code generation, code completion, code explanation, and code debugging, which require encoder or decoder type models [26–30].

CONCLUSION

In conclusion, the future development of AI coding systems holds great potential through execution-based feedback, reinforcement learning with human input, enhanced security analysis, and open-source collaboration. By integrating a runtime environment, these systems can validate outputs in real time, boosting reliability. Reinforcement Learning with Human Feedback will enable continual learning and personalization, while vulnerability detection can ensure robust and secure code. Finally, involving the open-source community will accelerate innovation, increase transparency, and expand the system's capabilities across diverse languages and use cases. Together, these advancements can significantly improve the accuracy, trustworthiness, and adaptability of AI-powered coding tools.

FUTURE SCOPE

Integration with IDEs and Online Coding Platforms

The fine-tuned model can be integrated into popular Integrated Development Environments (IDEs) such as VSCode, IntelliJ, or online platforms like LeetCode, HackerRank, and GitHub Copilot alternatives. This would enable real-time debugging, doubt solving, code generation and code completion for developers and learners.

Support for Multiple Programming Languages

Currently, the model may be fine-tuned for specific languages like Python or C. In the future, multilingual code support can be added by fine-tuning on diverse datasets across various languages (e.g., C++, Java, JavaScript, Go, etc.), which will make this capable to work for multiple languages.

Interactive Code Tutor System

The model can be extended to work as an interactive AI tutor that not only fixes bugs or explains code but also quizzes users and provides learning feedback to help them improve their coding skills. Various methods can be implemented to increase interactivity.

Incorporating Execution-Based Feedback

Future versions of the system could incorporate a runtime environment to execute generated/debugged code and ensure its correctness before presenting it to the user. This would improve reliability and trust and ensure that the resulted output is correct and working.

Reinforcement Learning and Human Feedback

The model can be further optimized using Reinforcement Learning with Human Feedback (RLHF), allowing it to learn from real users and adapt its responses based on developer preferences and code quality standards, which will make it updated over time on its own through user feedback.

Security and Vulnerability Detection

Expanding the model's capabilities to detect and suggest fixes for security issues and vulnerabilities in the code can be a highly impactful future direction. This will enhance the security and reduce chances of vulnerability, if any.

Open-Source Community Contributions

Opening the model and training pipeline to the open-source community can lead to faster innovation, broader language support, and real-world benchmarks.

REFERENCES

1. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155. 2020 Feb 19.
2. Wang Y, Wang W, Joty S, Hoi SC. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859. 2021 Sep 2.
3. Lachaux MA, Roziere B, Chanussot L, Lample G. Unsupervised translation of programming languages. arXiv preprint arXiv:2006.03511. 2020 Jun 5.
4. Chen M, Tworek J, Jun H, Yuan Q, Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. 2021 Jul 7.
5. Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q, Sutton C. Program synthesis with large language models. arXiv preprint arXiv:2108.07732. 2021 Aug 16.
6. Nijkamp E, Pang B, Hayashi H, Tu L, Wang H, Zhou Y, Savarese S, Xiong C. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474. 2022 Mar 25.
7. Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366. 2020 Sep 17.
8. Piskounova OI. Triple-Pomeron Diffraction Peak as a Signature of UHE Proton-Initiated Spectra of Gammas and Neutrinos in Astrophysics. arXiv preprint arXiv:2302.08546. 2023 Feb 15.
9. Zhang Q, Fang C, Ma Y, Sun W, Chen Z. A survey of learning-based automated program repair. ACM Trans Softw Eng Methodol. 2023 Dec 23; 33(2): 1–69.
10. Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, Li G. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664. 2021 Feb 9.
11. Shah D, Eysenbach B, Kahn G, Rhinehart N, Levine S. Rapid exploration for open-world navigation with latent goal models. arXiv preprint arXiv:2104.05859. 2021 Apr 12.
12. Li X, Liu S, Feng R, Meng G, Xie X, Chen K, Liu Y. Transrepair: Context-aware program repair for compilation errors. In Proceedings of the 37th IEEE/ACM international conference on automated software engineering. 2022 Oct 10; 1–13.
13. Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018: 1-12.
14. Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S. Language models are few-shot learners. Adv Neural Inf Process Syst. 2020; 33: 1877–901.
15. Sun Y, Wang S, Li Y, Feng S, Chen X, Zhang H, Tian X, Zhu D, Tian H, Wu H. Ernie: Enhanced representation through knowledge integration. arXiv preprint arXiv:1904.09223. 2019 Apr 19.
16. Lachaux MA, Roziere B, Chanussot L, Lample G. Unsupervised translation of programming languages. arXiv preprint arXiv:2006.03511. 2020 Jun 5.
17. Huang J, Gu SS, Hou L, Wu Y, Wang X, Yu H, Han J. Large language models can self-improve. arXiv preprint arXiv:2210.11610. 2022 Oct 20.

18. Nijkamp E, Pang B, Hayashi H, Tu L, Wang H, Zhou Y, Savarese S, Xiong C. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474. 2022 Mar 25.
19. Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, Li G. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664. 2021 Feb 9.
20. Fried D, Aghajanyan A, Lin J, Wang S, Wallace E, Shi F, Zhong R, Yih WT, Zettlemoyer L, Lewis M. Incoder: A generative model for code infilling and synthesis. arXiv preprint arXiv:2204.05999. 2022 Apr 12.
21. Li P, Yang J, Islam MA, Ren S. Making ai less' thirsty'. Commun ACM. 2025 Jul 1; 68(7): 54–61.
22. Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, Rodriguez A. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971. 2023 Feb 27.
23. Rogava J, Tsiklauri M, Vashakidze Z. On stability and convergence of a three-layer semi-discrete scheme for an abstract analogue of the Ball integro-differential equation. J Math Anal Appl. 2023 Feb 1; 518(1): 126664.
24. Svyatkovskiy A, Deng SK, Fu S, Sundaresan N. Intellicode compose: Code generation using transformer. In Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 2020 Nov 8; 1433–1443.
25. Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q, Sutton C. Program synthesis with large language models. arXiv preprint arXiv:2108.07732. 2021 Aug 16.
26. Ziegler DM, Stiennon N, Wu J, Brown TB, Radford A, Amodei D, Christiano P, Irving G. Fine-tuning language models from human preferences. arXiv preprint arXiv:1909.08593. 2019 Sep 18.
27. Chowdhery A, Narang S, Devlin J, Bosma M, Mishra G, Roberts A, Barham P, Chung HW, Sutton C, Gehrmann S, Schuh P. Palm: Scaling language modeling with pathways. J Mach Learn Res. 2023; 24(240): 1–13.
28. Mökander J, Schuett J, Kirk HR, Floridi L. Auditing large language models: a three-layered approach. AI Ethics. 2024 Nov; 4(4): 1085–115.
29. Colón KD, Kreidberg L, Welbanks L, Line MR, Madhusudhan N, Beatty T, Tamburo P, Stevenson KB, Mandell A, Rodriguez JE, Barclay T. An unusual transmission spectrum for the sub-saturn KELT-11b suggestive of a subsolar water abundance. Astron J. 2020 Nov 23; 160(6): 280.
30. Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. ACM Comput Surv. 2018 Jul 31; 51(4): 1–37.

