IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

Code Generation using Transformer Models

Prof. J. R. Mahajan[1] and Ms. Kedare Geetanjali[2]
Assistant Prof., Computer Engineering Department [1]
Students, M.E., Computer Engineering Department[2]
Adsul's Technical Campus. Chas, Ahilyanagar, India

Abstract: *Code generation, the task of automatically producing source code from natural language descriptions or partial code snippets, has seen significant advancements with the introduction of Transformer-based models. Unlike traditional rule-based or statistical methods, Transformer models leverage self-attention mechanisms and deep learning to better understand context, syntax, and semantics, thereby generating more accurate and human-like code. This paper explores the evolution of code generation, emphasizing the pivotal role of Transformer architectures such as GPT, Codex, and CodeT5. We discuss how these models are trained on massive code corpora, enabling them to perform tasks like code completion, translation between programming languages, and automatic bug fixing. Moreover, we highlight their application across various domains, from software development to educational tools, while analyzing the challenges including syntactic errors, logical inconsistencies, and ethical concerns like code plagiarism. The study also sheds light on ongoing research aimed at enhancing model efficiency, interpretability, and domain adaptability. Overall, Transformer models represent a transformative approach to automating coding tasks, holding great promise for the future of intelligent software engineering.*

Keywords: Code Translation, Transformer Models, Natural Language Processing, Code Generation, GPT Models, Code T5

I. INTRODUCTION

The automation of software development processes has been a longstanding goal within the field of Artificial Intelligence (AI). One key area in this pursuit is code generation -the automatic production of programming code from natural language instructions or incomplete code fragments. Transformer-based models, renowned for their success in Natural Language Processing (NLP), have revolutionized this domain by providing sophisticated, context-aware solutions. Their ability to model complex dependencies and generate coherent sequences makes them highly effective for coding tasks.

Traditional code generation methods, based on templates, grammar rules, or statistical models, often struggle with flexibility, scalability, and semantic understanding. With the advent of Transformer models, these limitations are significantly reduced. Models like Open AI's Codex and Salesforce's CodeT5 demonstrate how AI can now generate functionally correct, readable, and even optimized code. This shift opens new possibilities for faster development cycles, enhanced productivity, and democratized access to programming knowledge.

In recent years, code generation has emerged as a revolutionary field at the intersection of software engineering and artificial intelligence (AI). Specifically, Transformer models—originally designed for natural language processing (NLP) tasks—have demonstrated extraordinary capabilities in automatically generating high-quality code from natural language descriptions, partial code snippets, or problem statements. These models, built on attention mechanisms, are capable of understanding complex dependencies within sequences, making them ideal for tasks that require syntactic and semantic precision like code generation.

Transformer-based code generation operates by learning patterns, structures, and logic from massive corpora of programming languages. Models such as OpenAI's Codex (powering GitHub Copilot), CodeGen by Salesforce, PolyCoder, and Google's AlphaCode have set new benchmarks in this domain. These models are often trained on

DOI: 10.48175/568

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

datasets like The Pile, CodeSearchNet, and GitHub repositories, enabling them to generate coherent, functional code across multiple languages, including Python, Java, C++, JavaScript, and more.

However, Transformer-based code generation is not without challenges. Issues such as hallucinated code (where the model produces syntactically correct but logically incorrect code), security vulnerabilities, license compliance, and bias toward frequently seen patterns pose serious concerns. As a result, significant research is being conducted to improve model safety, interpretability, and controllability.

In practical applications, Transformer models are being integrated into Integrated Development Environments (IDEs), code review tools, and software testing platforms. They help developers by boosting productivity, reducing boilerplate code, and even aiding non-programmers in creating simple applications via low-code/no-code platforms. In educational contexts, they provide instant feedback to programming students and generate exercises tailored to learning goals.

In summary, code generation using Transformer models represents a significant technological advancement with profound impacts on software development, education, and AI research. With ongoing innovations and ethical considerations, it is poised to redefine how humans and machines collaborate in programming tasks.

II. LITERATURE SURVEY

For our project we are surveying some reports and references which are helping us to make it easy and simplest and they are as follows

1. Enrique Dehaerne et al,“ Code Generation Using Machine Learning : A Systematic Review ”

In this they study generating code of programming language for automatic software development. This review provides a broad and detailed over view of studies for code generation using Machine Learning. The most popular applications that works in these paradigms is code generation from natural language descriptions, documentation generation, and automatic program repair. The most frequently used Machine Learning model in these studies include Transformer and Recurrent Neural Network and Conventional Neural Network. Other Neural Network architecture as well as non-neural techniques were also observed.

2. Youngmi et al“ALSI-Transformer: Transformer-Based Code Comment Generation with Aligned Lexical and Syntactic Information”

Code comments explain the operational process of a computer program and increase the long term productivity of programming tasks such as debugging and maintenance. Therefore, developing methods that automatically generate natural language comments from programming code is required. With the development of deep learning, various excellent models in the natural language processing domain have been applied for comment generation tasks, and recent studies have improved performance by simultaneously using the lexical information of the code token and the syntactical information obtained from the syntax tree. In this paper, to improve the accuracy of automatic comment generation, we introduce a novel syntactic sequence, Code-Aligned Type sequence (CAT), to align the order and length of lexical and syntactic information, and we propose a new neural network model, Aligned Lexical and Syntactic information-Transformer (ALSI-Transformer), based on a transformer that encodes the aligned multi-modal information with convolution and embedding aggregation layers. Through in-depth experiments. They compared ALSI Transformer with current baseline methods using standard machine translation metrics and demonstrate that the proposed method achieves state-of-the-art performance in code comment generation.

3. Das N et al“A Comparative Study on Code Generation with Transformers”

In an era of widespread influence of Natural Language Processing (NLP), there have been multiple research efforts to supplant traditional manual coding techniques with automated systems capable of generating solutions autonomously. With rapid research for code generation and a sole focus on large language models, there emerges a need to compare and evaluate the performance of transformer architectures based on several complexities of the model. This paper introduces the concept of “A Comparative Study on Code Generation with Transformers,” a model based on Transformer architecture and NLP methodologies to automatically generate C++ source code for different varieties of problems. Here, a comparative study is performed to evaluate the robustness of transformer-based models on the basis

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

of their architecture complexities and their capability to handle diverse problem sets, from basic arithmetic to complex computations.

4. Mark Chen,Jerry T et al"Evaluating Large Language Models Trained on Code"
In this paper they introduce Codex, a GPT language model fine-tuned on publicly available code from GitHub, and study its Python code-writing capabilities. A distinct production version of Codex powers GitHub Copilot. On Humane Val, a new evaluation set we release to measure functional correctness for synthesizing programs from doc strings, our model solves 28.8% of the problems, while GPT-3 solves 0% and GPT-J solves 11.4%. Furthermore, they find that repeated sampling from the model is a surprisingly effective strategy for producing working solutions to difficult prompts.

5. Wang Y et al"Code T5: Identifier aware Unified Pre-Trained Encoder-Decoder Models for code Understanding and Generation"
Pre-trained models for Natural Languages (NL) like BERT and GPT have been recently shown to transfer well to Programming Languages (PL) and largely benefit a broad set of code-related tasks. Despite their success, most current methods either rely on an encoder-only (or decoder-only) pre-training that is suboptimal for generation (resp. understanding) tasks or process the code snippet in the same way as NL, neglecting the special characteristics of PL such as token types. We present CodeT5, a unified pre-trained encoder-decoder Trans- former model that better leverages the code semantics conveyed from the developer-assigned identifiers. Our model employs a unified framework to seamlessly support both code understanding and generation tasks and allows for multi-task learning.

III. SYSTEM DESIGN

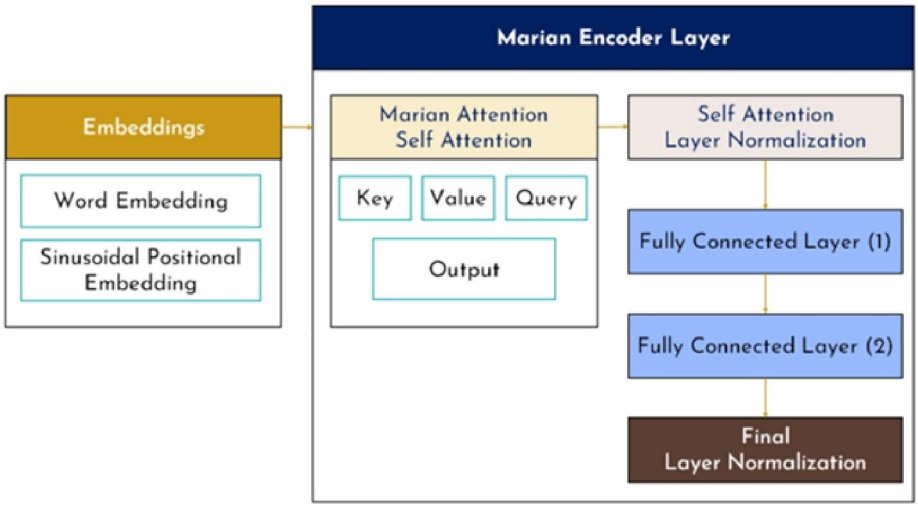

Fig. 1. System Architecture

The system architecture for code generation using Transformer models typically consists of the following stages:

A. Input Module:

• Accepts user inputs such as natural language prompts, incomplete code snippets, or function descriptions.
• Pre-processing like tokenization is performed here.

B. ENCODER (TRANSFORMER-BASED):

• Encodes the input prompt into a dense, context-rich representation.
• Captures syntactic and semantic information from the input.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

C. DECODER (TRANSFORMER-BASED):
• Generates code tokens sequentially based on the encoded input and previously generated tokens.
• Predicts the next most probable code element.

D. TRAINING PHASE (OFFLINE):
• The Transformer model is trained on large code corpora (e.g., GitHub, Stack Overflow datasets).
• Loss functions like Cross-Entropy or CodeBLEU are used for optimization.

E. INFERENCE/GENERATION PHASE (ONLINE):
• The trained model generates code snippets in real-time for new inputs.
• Post-processing ensures syntax correctness and format adjustments.

F. EVALUATION MODULE:
• Validates the generated code through syntactic checks, test cases, or human evaluation.
• Provides feedback for fine-tuning if necessary.

G. OUTPUT MODULE:
• Displays the final generated code to the user.
• May also provide alternative suggestions or comments.
Automatic Code Generator using a Transformer Model: Component Descriptions. This architecture outlines a common pipeline for systems that automatically generate code from natural language descriptions, heavily relying on the power of Transformer networks.

IV. METHODOLOGY

Module Description
The methodology for Code Generation Using Transformer Models involves a systematic sequence of stages that span from data preparation to model deployment. The central objective is to train a Transformer-based system capable of translating user-provided natural language descriptions into accurate, syntactically correct, and semantically meaningful source code. The following steps outline the core processes involved.

A. Data Collection and Preparation
The foundation of effective code generation lies in the availability of large, diverse, and high-quality datasets. Code repositories such as GitHub, CodeSearchNet, and The Pile serve as primary sources. These datasets include millions of code snippets, functions, and complete programs across multiple programming languages.
Before training, the raw data undergoes preprocessing steps:

- Deduplication to remove redundant code samples.
- Cleaning to eliminate incomplete or corrupted files.
- Filtering based on licensing requirements to ensure ethical use of open-source code.
- Annotation (optional) where pairs of natural language descriptions and corresponding code snippets are curated for supervised learning.

B. Preprocessing
Both the natural language prompts and the code samples must be transformed into a format suitable for model ingestion.
• Tokenization
Splits text and code into smaller units (tokens) using specialized tokenizers that handle programming language syntax (e.g., handling brackets, commas, operators).

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

• Encoding converts these tokens into vector representations (embeddings) that capture semantic and syntactic information.

Specialized tokenization methods like Byte Pair Encoding (BPE) or SentencePiece are often used to handle both text and code uniformly.

C. Model Selection & architecture

The core engine of the system is a Transformer model. Two main types of architectures can be used.

Encoder-Decoder Transformers (e.g., T5, CodeT5).

Useful when both understanding the input and generating structured output is crucial.

Decoder-Only Transformers (e.g., GPT-2, Codex).

Focused on autoregressive generation, predicting the next token based on previous tokens.

The model incorporates:

Self-Attention Mechanisms to understand relationships within the input and output sequences.

Cross-Attention Mechanisms (in encoder-decoder models) to align the input text with the generated code sequence.

Positional Encodings to retain the order of tokens, essential for programming syntax.

D. Training Phase

The model is trained using large batches of tokenized input-output pairs:

- Objective Function: Commonly, the Cross-Entropy Loss is used to measure the difference between the predicted tokens and the actual tokens.
- Optimization Algorithms: Advanced optimizers like Adam or AdamW are utilized to update model weights efficiently.

Regularization Techniques: Methods such as dropout, weight decay, and label smoothing are employed to prevent overfitting Depending on the task, the training can be:

- Supervised Learning (with labeled prompt-code pairs)
- Self-Supervised Learning (predicting masked or missing tokens within code)

Transfer learning and fine-tuning strategies are also applied, where a pretrained language model is adapted to code-specific tasks.

E. Interface & Code Generation

- The user provides a natural language prompt.
- The model tokenizes and encodes the input.
- Based on the prompt, the model generates code token-by-token using techniques like:

o Greedy Search (selecting the most probable token at each step)

o Beam Search (considering multiple possible sequences)

o Top-k Sampling or Nucleus Sampling (introducing randomness for more diverse outputs)

F. Post Processing

Post processing ensures the usability and correctness of the generated code:

- Syntax Checking: Automatic validation against the grammar rules of the target programming language.
- Formatting: Applying standard code formatting styles (e.g., PEP8 for Python).
- Execution Testing (optional): Running the generated code in sandbox environments to verify functional correctness.

G. Evaluation & Validation

The performance of the code generation system is evaluated using various metrics:

- Exact Match Accuracy (whether the generated code exactly matches the expected output)

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

- CodeBLEU Score (measures n-gram match, syntax match, and data-flow match)
- Functional Correctness (whether the generated code performs the intended task)

H. Deployment

After satisfactory evaluation, the trained model can be deployed into real-world applications:

- Integration into IDEs (e.g., VS Code, PyCharm)
- Web-based code assistants
- APIs for backend integration
- Educational tools for teaching programming

Security measures, model monitoring, and user feedback loops are also implemented to continually improve system performance after deployment.

V. CONCLUSION

Transformer-based models have significantly transformed the landscape of automatic code generation, offering a paradigm shift from traditional rule-based or statistical methods to deep learning-driven, context-aware generation. Their ability to model long-range dependencies, understand sequence semantics, and capture contextual relationships has made them particularly suitable for complex tasks in programming language understanding and generation.

Through extensive training on massive code corpora and natural language documentation, these models can now generate code snippets, complete functions, translate between languages, and even write basic applications with minimal human intervention. Tools like OpenAI Codex and GitHub Copilot exemplify this evolution, enabling developers to work more efficiently, reduce cognitive load, and automate repetitive programming tasks.

Moreover, Transformer models are playing a pivotal role in redefining human-AI collaboration in software engineering. They are not just limited to code synthesis; they support intelligent code completion, suggest bug fixes, generate tests, and even provide documentation—all contributing to a more seamless and productive software development lifecycle. The integration of these models within IDEs and development pipelines is also helping democratize coding, making it more accessible to non-programmers and novice learners.

VI. FUTURE SCOPE

- Development of domain-specific transformer models (e.g., for finance, robotics).
- Enhancing logical reasoning and debugging capabilities in generated code.
- Integrating with IDEs for real-time code optimization and security alerts.
- Supporting multimodal inputs like diagrams or voice instructions.
- Legal and ethical research on intellectual property of generated code.

ACKNOWLEDGMENT

It gives us great pleasure in presenting the paper on " Code Generation Using Transformer Models". We would like to take this opportunity to thank our guide, Prof. J. R. Mahajan, Professor, Department of Computer Engineering, Adsul's Technical Campus , Chas, for giving us all the help and guidance we needed. We are grateful to her for her kind support, and valuable suggestions were very helpful.

REFERENCES

[1]. Enrique Dehaerne,Bappaditya Dey,Sandip Halder,Stefan De Gendt,Wannes Meert, " Code Generation Using Machine Learning : A Systematic Review " IEEE Access 2022

[2]. Youngmi Park,Ahjeong Park,Chulyun Kim "ALSI-Transformer: Transformer-Based Code Comment Generation With Aligned Lexical and Syntactic Information " IEEE Access 2023

[3]. Das Namrata,Rakshay Pant,Neelam Karki,Ruchi Manandhar,Dinesh Baniya Kshatri"A Comparative Study on Code Generation with Transformers" arXIV 2024

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

[4]. Mark Chen, Jerry Tetal "Evaluating Large Language Models Trained on Code" arXiv2021

[5]. Wang Yue,W Wang, SJoty,Steven C"Code T5: Identifier –aware Unified Pre-Trained Encoder-Decoder Models for code Understanding and Generation" arXiv2021

[6]. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou,Bing Qin, Ting Liu, Daxin Jiang, Ming Zhou "CodeBERT:A Pre-Trained Model for Programming and Natural Languages"arXiv2020

[7]. Hamel Husain ,Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis Marc B "Code Search Net Challenge:Evaluating the Semantic Code Search "arXir2020

[8]. Vaswani, A., et al. "Attention is All You Need." NeurIPS, 2017

DOI: 10.48175/568