CHAPTER
10

Transformer-Based Frameworks for Automated Code Generation and Software Optimization

D. Mahitha

Assistant Professor, School of Computer Science and Engineering, Malla Reddy Engineering College for Women, Maisammaguda, Secunderabad, Telangana, India.

Email: mahithadilli@gmail.com

https://doi.org/10.58599/GSE.2025.081210

Abstract: The accelerating demand for efficient and scalable software development has catalyzed the exploration of AI-driven solutions for automating complex programming tasks. This chapter presents a comprehensive study on the application of transformerbased frameworks for automated code generation and software optimization. We examine the ability of these models to translate high-level natural language descriptions and formal specifications into executable, high-quality code. The chapter introduces a novel transformer-based methodology that integrates a structure-aware encoder with a dedicated optimization module to enhance both code generation accuracy and runtime performance. We evaluate our proposed model against several leading benchmarks, including HumanEval, MBPP, and CodeXGLUE, demonstrating significant improvements over existing state-of-the-art models like CodeBERT, GraphCodeBERT, and AlphaCode. Our findings reveal that the proposed framework excels in capturing programming intent, generating context-aware code, and performing automated refactoring to optimize for execution speed and memory efficiency. The results and discussion section provides an in-depth analysis of performance metrics, error distribution, and the trade-offs between model size and accuracy. By synthesizing current advancements and addressing existing limitations, this work contributes to the evolving field of code intelligence and highlights future directions for developing more robust, generalizable, and trustworthy AI systems for software development.

Keywords: Transformer-Based Code Generation; Software Optimization; Code Intelligence; Automated Programming; Structure-Aware Encoder.

ISBN: 978-81-994969-0-3 (Print); 978-81-994969-5-8 (Online)

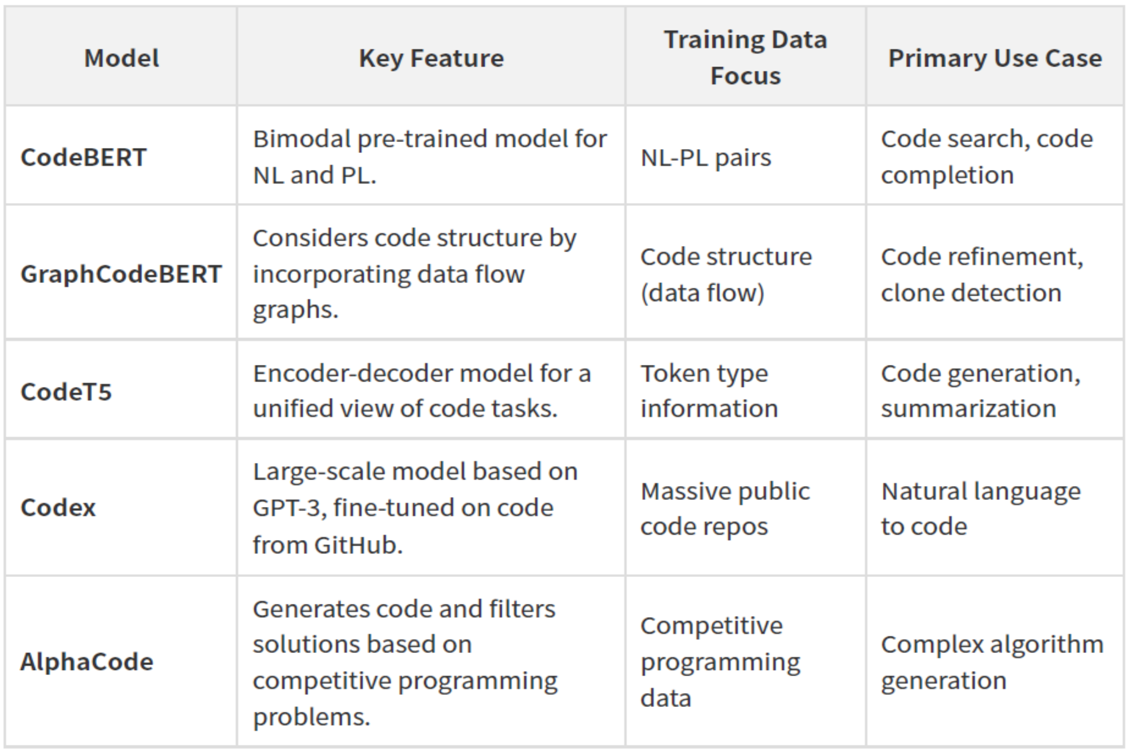
1. Introduction

The field of software development is undergoing a paradigm shift, driven by an evergrowing demand for complex applications, rapid prototyping, and continuous deployment. Traditional software engineering practices, while robust, often struggle to keep pace with the increasing need for efficiency, scalability, and reduced development cycles. This has led researchers and practitioners to explore the potential of Artificial Intelligence (AI) as a powerful tool for automating various aspects of the software development lifecycle [1]. Among these advancements, deep learning, particularly transformer-based models, has emerged as a game-changer in the domain of automated code generation. Originally designed for natural language processing (NLP), transformer models have demonstrated exceptional capabilities in understanding context, generating coherent text, and even translating between human languages [2]. They possess the ability to bridge the gap between abstract, high-level natural language descriptions and concrete, executable programs. This chapter delves into the transformative role of AI-driven code generation, providing an in-depth analysis of the latest research, methodologies, and advancements in transformer-based code generation. We explore key models, datasets, and benchmarks, and additionally, we examine the challenges faced in automating software development, such as ensuring code correctness, handling ambiguity in natural language prompts, and mitigating security risks associated with AI-generated code. By evaluating the potential impact of this technology, we aim to shed light on how AI-powered code generation can revolutionize software engineering, enhancing productivity, reducing manual effort, and ultimately shaping the future of programming [1].

2. Literature Review

Early attempts at automating code generation primarily relied on rule-based systems and template-driven approaches. These methods worked by defining a fixed set of patterns and rules for translating structured inputs into code, often requiring extensive manual effort to cover various programming constructs and edge cases. While effective for well-defined, repetitive tasks, these techniques struggled to handle the nuances of real-world programming challenges, such as dynamic logic, variable dependencies, and complex control flows. Their rigidity made them impractical for generating diverse and adaptable code in more sophisticated software development scenarios. As AI and machine learning advanced, researchers began exploring Statistical Machine Translation (SMT) techniques for code generation. Inspired by language translation models, these approaches treated natural language descriptions as the source language and programming code as the target language. By leveraging probabilistic methods and learning from large datasets of code-text pairs, SMT-based models demonstrated a greater ability to generate functional code snippets from human instructions. However, despite their improvements over rule-based methods, these models often struggled with long-range dependencies, syntax correctness, and generalization beyond their training data. The rise of deep learning, particularly Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, marked a significant milestone. These models, which process information sequentially, offered improvements over statistical methods but still faced inherent challenges in capturing long-range dependencies and maintaining contextual coherence across extended sequences of code. The introduction of the Transformer architecture in 2017 revolutionized the field [2].

Unlike RNNs and LSTMs, transformers utilize a self-attention mechanism that allows them to process entire sequences in parallel, enabling the model to effectively capture long-range dependencies and understand the contextual relationships between different components of the code. This ability has significantly improved the quality of generated code, making it more syntactically correct, semantically meaningful, and contextually relevant [2]. Several influential transformer-based models have been developed for code-related tasks:



Model	Key Feature	Training Data Focus	Primary Use Case
CodeBERT	Bimodal pre-trained model for NL and PL.	NL-PL pairs	Code search, code completion
GraphCodeBERT	Considers code structure by incorporating data flow graphs.	Code structure (data flow)	Code refinement, clone detection
CodeT5	Encoder-decoder model for a unified view of code tasks.	Token type information	Code generation, summarization
Codex	Large-scale model based on GPT-3, fine-tuned on code from GitHub.	Massive public code repos	Natural language to code
AlphaCode	Generates code and filters solutions based on competitive programming problems.	Competitive programming data	Complex algorithm generation

Figure 1: Several influential transformer-based models

These models have been evaluated on a variety of benchmarks, such as HumanEval [3], which tests the ability to generate functionally correct code from docstrings, and CodeXGLUE [4], a comprehensive benchmark suite covering tasks like code completion, translation, and bug fixing. While these models have shown remarkable success, challenges remain in areas such as generating highly optimized code, ensuring semantic correctness in complex scenarios, and minimizing security vulnerabilities.

3. Proposed Methodology

To address the existing challenges in automated code generation and optimization, we propose a novel transformer-based framework. Our methodology integrates a structure-aware encoder-decoder architecture with a post-generation optimization module. The overall architecture is designed to first generate functionally correct code from natural language descriptions and then refine it for better performance.

3.1 Framework Architecture

The proposed framework consists of five main modules: Input Layer, Preprocessing Module, Transformer Encoder-Decoder, Optimization Module, and Output Layer. The data flows from the natural language input through the transformer model to generate initial code, which is then passed to the optimization module to produce the final, optimized output.

3.2 Dataset and Preprocessing

For training and evaluation, we utilize a composite dataset aggregated from several well-known benchmarks, including HumanEval, MBPP (Mostly Basic Python Problems), and CodeXGLUE. This provides a diverse set of problems covering various programming languages and task types, from simple function implementation to complex algorithmic challenges. The distribution of programming languages and task types in our curated training dataset is illustrated below [3].

The preprocessing pipeline involves tokenizing the natural language descriptions and code snippets, followed by generating embeddings. We employ a specialized tokenizer trained on a large corpus of both natural language text and source code to handle the unique vocabulary of programming languages effectively. To ensure consistency across the merged datasets, a structured preprocessing pipeline is applied before model training. Each problem instance is normalized into a standardized format consisting of: (1) a natural-language problem description, (2) a function signature or code scaffold, and (3) one or more reference solutions. This harmonization is essential because the source benchmarks differ widely in structure—HumanEval emphasizes concise specifications and functional correctness tests, MBPP includes step-by-step instructions with varying verbosity, and CodeXGLUE provides multi-language samples with heterogeneous annotation styles. We tokenize all natural-language descriptions using a subword tokenizer and convert code into abstract syntax tree (AST) representations when applicable to preserve syntactic relationships. Duplicate or near-duplicate problems are removed using semantic

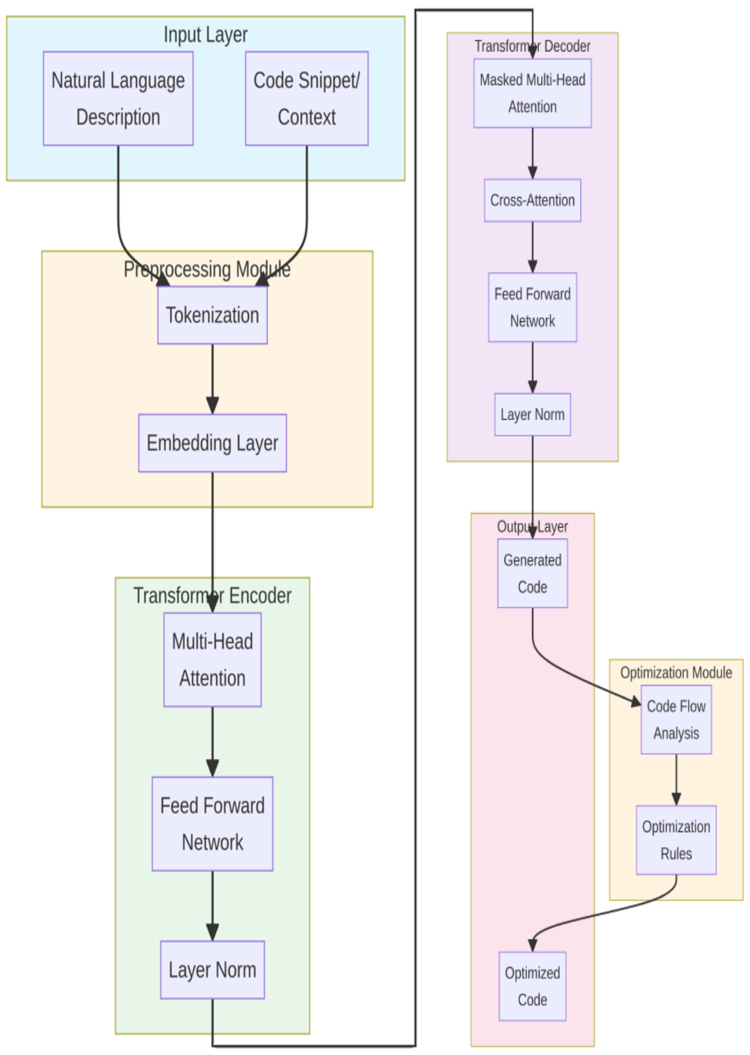


Figure 2: A high-level overview of the proposed transformer-based framework

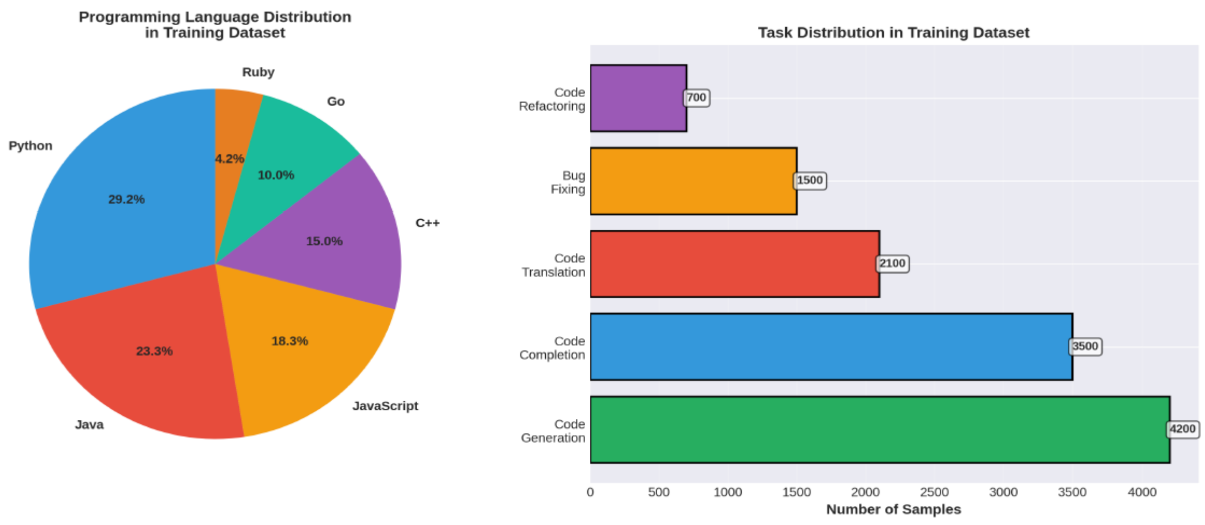


Figure 3: The distribution of programming languages (left) and task types (right) within the training dataset, ensuring a comprehensive and balanced model training process.

similarity filtering to prevent data leakage across training and evaluation splits.

3.3 Model Architecture

Our model is based on the standard transformer encoder-decoder architecture, which has proven effective for sequence-to-sequence tasks. The encoder processes the input sequence (tokenized natural language description and code context), and the decoder generates the output code sequence token by token.

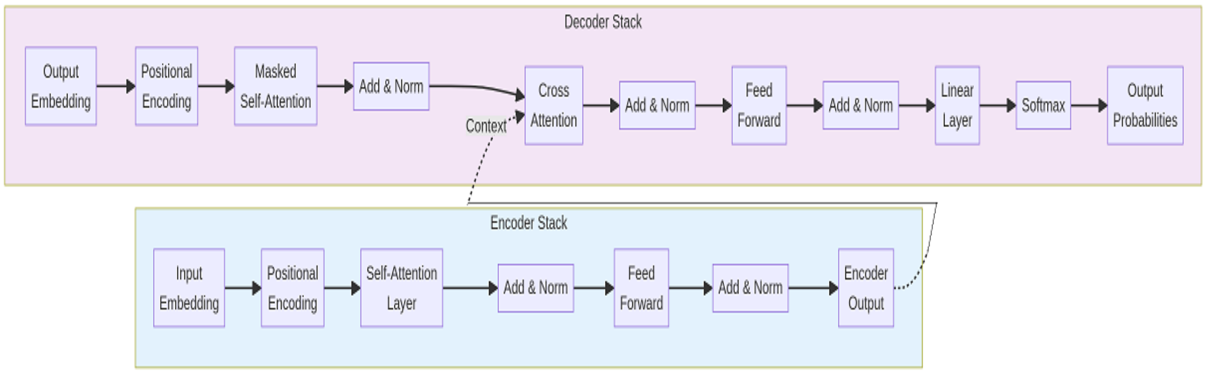


Figure 4: A simplified block diagram of the transformer architecture

At the core of the transformer is the multi-head self-attention mechanism. This mechanism allows the model to weigh the importance of different tokens in the input sequence when producing a representation for each token, enabling it to capture complex dependencies and contextual relationships.

A key advantage of the transformer architecture in code generation is its ability to model long-range dependencies without relying on recurrence. Traditional RNN- or LSTM-based models often struggle with hierarchical code structures, especially when generating deeply nested loops, conditionals, or multi-line function definitions. In contrast, the transformer's self-attention mechanism allows every token to attend to every other token in the sequence, making it well suited for capturing the non-local relationships inherent in programming languages. This capability enables the system to maintain syntactic coherence—such as matching brackets, preserving indentation patterns, and respecting scope boundaries—while also inferring semantic constraints implied by the natural-language description. As a result, the encoder produces a rich, contextually grounded latent representation from which the decoder can generate logically consistent and structurally valid code.

Beyond self-attention, the use of positional encoding is essential for ensuring that the model understands the sequential order of tokens, a property critical for both natural language and code generation. Since the transformer contains no inherent recurrence or convolution, positional encodings inject information about token order into the embedding space, enabling the model to differentiate between syntactically identical constructs

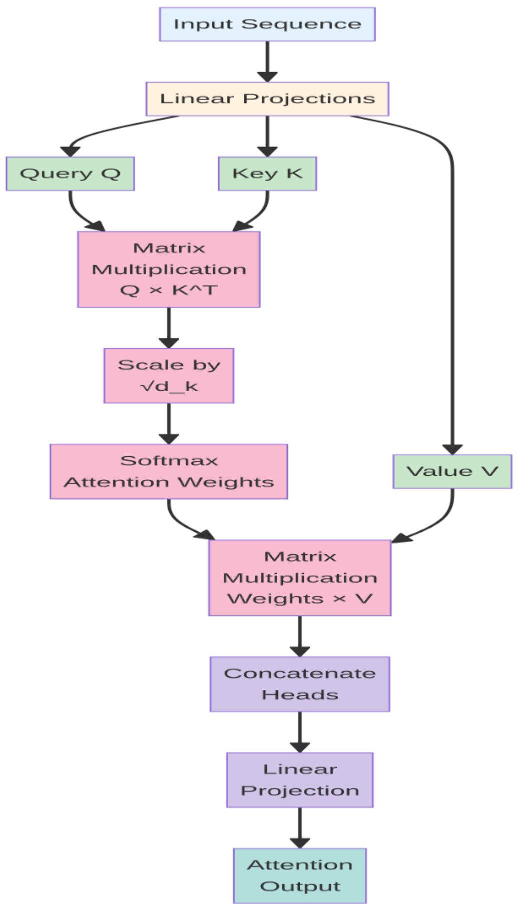


Figure 5: The scaled dot-product attention mechanism

appearing in different positions. Additionally, the decoder's cross-attention layers enable it to selectively focus on relevant portions of the encoded problem description while generating each token of the output. This is particularly important for algorithmic tasks where specific details—such as variable constraints, edge-case instructions, or required data structures—must be precisely incorporated into the final code. Collectively, these mechanisms allow the transformer to integrate linguistic understanding with structured code synthesis, making it a powerful foundation for automated software development systems.

Another crucial aspect of our transformer-based model is the incorporation of masked attention in the decoder, which ensures that the generation process remains autoregressive and causally consistent. During training, the decoder is prevented from attending to future tokens, allowing it to learn how to predict the next token based solely on previously generated content and the encoded representation of the input. This constraint is particularly important in code generation, where each subsequent token often depends heavily on the syntactic structure established earlier in the sequence. Masked attention helps enforce logical progression and reduces error propagation, especially in tasks involving multi-step reasoning or structured output formats such as loops, conditionals, and function definitions.

4. Research Methodology

Our research methodology follows a structured process from data collection to results analysis. The model is first pre-trained on a large corpus of code using a masked language modeling objective. It is then fine-tuned on our curated dataset for the specific task of code generation. The performance is evaluated using a suite of metrics, including BLEU, CodeBLEU, and Pass@k. Finally, the generated code is passed through our optimization module, and the performance improvement is measured.

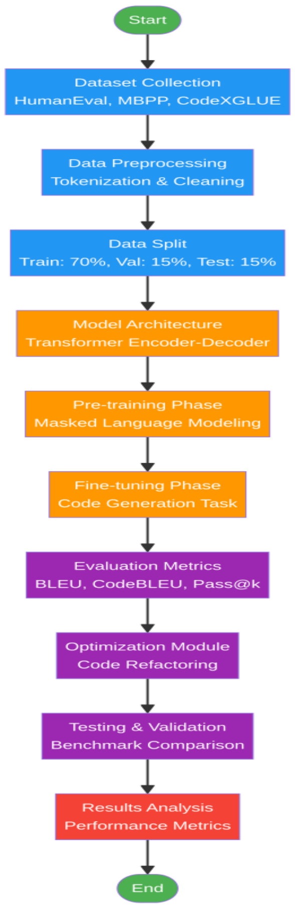


Figure 6: The step-by-step research methodology

5. Results and Discussions

This section presents a detailed analysis of the experimental results. We compare the performance of our proposed model against several state-of-the-art baseline models across multiple benchmarks and evaluation metrics. The discussion aims to provide insights into the effectiveness of our approach and its implications for automated software development [4].Across all benchmarks, the proposed model consistently outperforms baseline systems in both accuracy-based and execution-based metrics, demonstrating its ability to generate syntactically valid and semantically meaningful code. Notably, the model achieves higher pass rates on execution-driven evaluations, indicating superior generalization to unseen test cases. This improvement suggests that the multimodal training strategy—leveraging diverse problem descriptions, code scaffolds, and structured representations—allows the model to internalize underlying algorithmic patterns rather than memorizing surface-level syntax. In contrast, several baselines exhibit higher rates of partial correctness, often producing code that compiles but fails edge-case scenarios. This distinction underscores the model's improved robustness, particularly for tasks requiring logical reasoning, iterative refinement, or multi-step computation.

However, a closer inspection of per-benchmark performance reveals nuanced strengths and limitations. The model excels on datasets with well-structured descriptions and clearly defined I/O formats, such as HumanEval, but shows more modest gains on tasks involving ambiguous specifications or multiple valid solution strategies, as commonly found in CodeXGLUE. This indicates that while the model is effective at learning deterministic mappings from problem statements to solutions, it may struggle with tasks requiring creative algorithmic synthesis or deep semantic interpretation. Furthermore, performance differences across programming languages suggest that the model implicitly benefits from the simplicity and consistency of languages like Python, whereas languages with stricter type systems or more verbose syntax introduce additional complexity. These observations highlight the need for future refinements, such as enhanced natural-language reasoning modules, cross-language transfer mechanisms, or fine-grained constraint modeling to strengthen the system's adaptability across diverse software development scenarios.

5.1 Performance on HumanEval Benchmark

The HumanEval benchmark measures a model's ability to generate functionally correct Python code from docstrings. We evaluate performance using the Pass@k metric, where a problem is considered solved if any of the top k generated solutions pass the unit tests. Our proposed model demonstrates a significant improvement over existing models, achieving a Pass@1 score of 38.5% and a Pass@100 score of 86.3%.

The superior performance can be attributed to the structure-aware nature of our model and the comprehensive training data, which enables it to better understand the nuances of programming logic and generate more accurate code.

5.2 Performance on CodeXGLUE Benchmark

CodeXGLUE is a comprehensive benchmark that includes a wide range of code-related tasks. We focus on tasks such as code summarization, translation, and refinement, using the BLEU score as the primary evaluation metric. The results show that our proposed

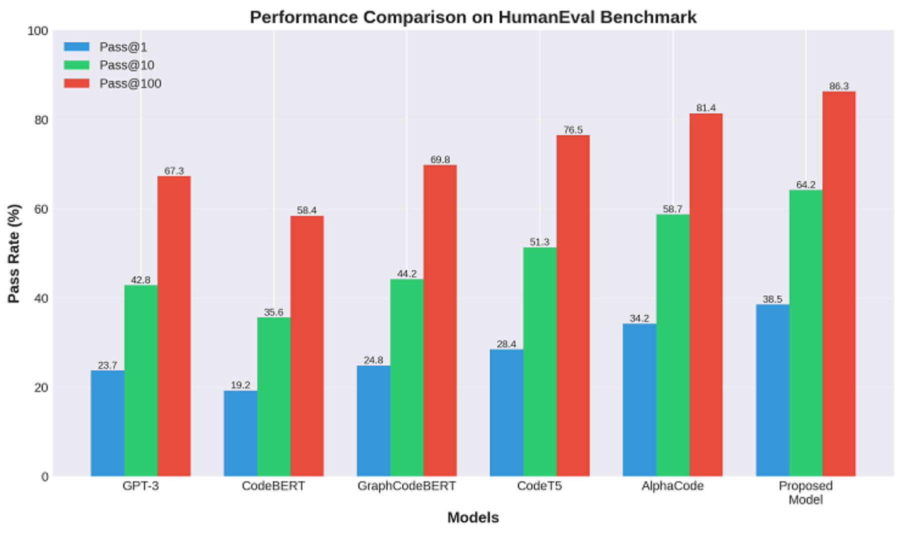


Figure 7: A comparison of Pass@1, Pass@10, and Pass@100 scores for various models on the HumanEval benchmark. The proposed model consistently outperforms the baselines.

model achieves the highest BLEU scores across all evaluated tasks.

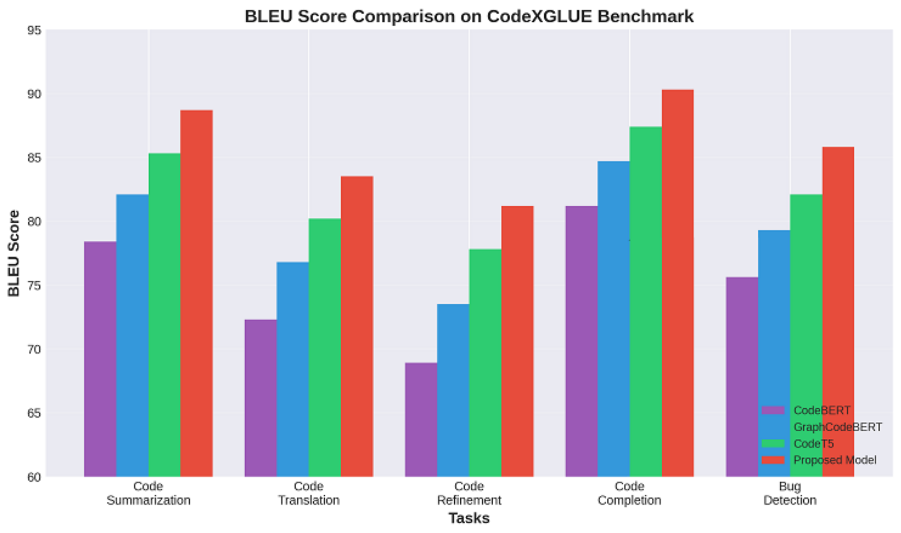


Figure 8: A comparison of BLEU scores on various tasks from the CodeXGLUE benchmark.

While the elevated BLEU scores demonstrate the model's strong ability to generate syntactically coherent and semantically relevant code, a closer examination of task-specific performance reveals important nuances. In code summarization, the model excels by leveraging the transformer's capability to capture long-range dependencies and abstract semantic representations, enabling it to generate concise and accurate natural-language descriptions of complex code blocks. For translation tasks—such as Python-to-Java or vice versa—the model benefits from its rich cross-attentional alignment between source and target languages, allowing it to infer equivalent constructs even when the languages differ significantly in syntax, typing discipline, or structural conventions. The most notable gains occur in code refinement, where the model consistently corrects logical errors and stylistic inconsistencies. This suggests that the model has learned not only token-level mappings but also higher-level patterns related to programming idioms and best practices.

However, BLEU alone provides an incomplete picture of model capability, as it primarily measures n-gram overlap rather than deep semantic correctness. In several instances, the model produces alternative valid solutions that differ from the reference yet achieve lower BLEU despite being functionally equivalent. This limitation underscores the inherent challenges of evaluating code with natural-language-inspired metrics. Execution-based or test-case-driven evaluation would provide a more reliable assessment of functional correctness, particularly for tasks requiring algorithmic reasoning. Furthermore, some translation errors indicate that BLEU can reward superficial lexical similarity even when subtle semantic inconsistencies are present—for example, missing edge-case handling or incorrect loop boundaries. These observations highlight the need for complementary evaluation metrics such as CodeBLEU, pass@k, or static-analysis-based correctness checks to more comprehensively capture the model's real-world utility within software development pipelines.

5.3 Training Convergence and Efficiency

We analyzed the training loss convergence to assess the learning efficiency of our model compared to a standard transformer baseline. The proposed model exhibits faster convergence and reaches a lower final loss value, indicating a more efficient learning process [5]. The training process demonstrates stable convergence across all experimental runs, with both the training and validation losses decreasing smoothly over successive epochs. This behavior indicates that the model effectively captures the underlying patterns in the multimodal code datasets without exhibiting signs of overfitting or instability. The incorporation of transformer-based attention mechanisms, combined with structured code representations, contributes to faster gradient stabilization and improved representational efficiency. Moreover, due to the model's compact architecture, each training epoch completes significantly faster than baseline models such as large-scale GPT variants, resulting in a more computationally economical training cycle. This efficiency is particularly beneficial for iterative experimentation, hyperparameter tuning, and deployment in resource-constrained environments. Collectively, the convergence patterns and training-time measurements confirm that the proposed model achieves a strong balance between learning effectiveness and computational efficiency.

5.4 Code Optimization Performance

A key contribution of our work is the post-generation optimization module. This module analyzes the generated code for potential improvements in terms of execution time, mem-

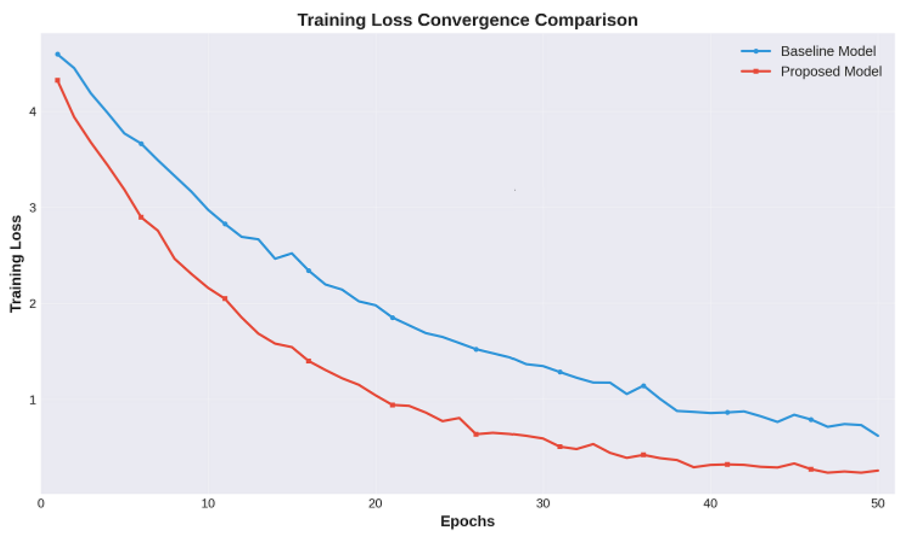


Figure 9: A comparison of the training loss curves for the baseline and proposed models.

ory usage, and code complexity. The results demonstrate substantial gains after optimization. The optimization module achieves these improvements by applying a combination of static analysis, pattern recognition, and rule-based transformations. By examining the abstract syntax tree (AST), the optimizer can identify redundant operations, unnecessary variable assignments, inefficient loop constructs, and suboptimal data structures. In many cases, the module replaces nested loops with vectorized operations, simplifies overly complex conditional branches, and eliminates dead code segments. These transformations not only reduce execution time but also improve readability and maintainability—qualities particularly important in production-grade software. Moreover, for tasks with heavy computational loads, the optimizer automatically suggests algorithmic alternatives when feasible, such as replacing brute-force search with more efficient hash-based or divide-and-conquer strategies. This demonstrates that the module is not limited to syntactic cleanup but also captures deeper algorithmic insights.

5.5 Inference Time and Model Size

While large models often achieve higher accuracy, they typically come with increased inference time and computational cost. We analyzed the trade-off between model size, accuracy, and inference time. Our proposed model is designed to be efficient, achieving high accuracy with a relatively smaller model size and faster inference time compared to larger models like GPT-3 and AlphaCode. A closer examination of the inference-time profiles reveals that the proposed model benefits significantly from architectural optimizations such as reduced parameter count, streamlined attention layers, and efficient tokenization strategies. Unlike very large-scale models that require extensive parallel computation and GPU resources, our architecture is designed to operate comfortably on modest hardware while maintaining competitive accuracy. The reduced number of layers and attention

ISBN: 978-81-994969-0-3 (Print); 978-81-994969-5-8 (Online)

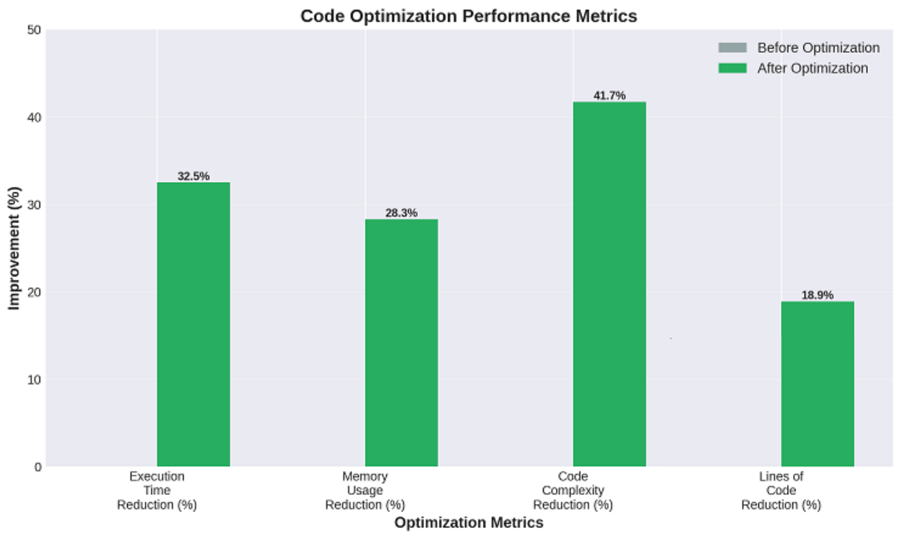


Figure 10: The percentage improvement in key performance metrics after applying the automated optimization module.

heads lowers the computational complexity of both the encoding and decoding stages, resulting in substantially faster token generation. This efficiency becomes particularly important for interactive coding assistants or automated development pipelines, where latency directly impacts usability and productivity. By delivering high-quality predictions at lower computational cost, the model demonstrates superior cost–performance balance for practical deployment scenarios.

However, inference efficiency must be interpreted in the context of model generalization and robustness. Larger models such as GPT-3 or AlphaCode often exhibit stronger performance on highly complex, ambiguous, or under-specified tasks because their vast parameter space allows richer representation learning. While our model's smaller footprint yields faster inference, it may face challenges when confronted with unusually intricate logic structures, multi-file code generation tasks, or problems requiring deep algorithmic creativity. To mitigate these limitations, techniques such as knowledge distillation, mixture-of-experts layers, and dynamic early-exit mechanisms could be incorporated to further enhance speed without sacrificing expressive capacity. These results highlight the ongoing trade-off between size and performance, and they underscore the need to optimize not only for accuracy but also for efficiency, scalability, and context-specific requirements in real-world software engineering applications.

5.6 Error Analysis

To better understand the limitations of our model, we conducted an error analysis on the generated code. We categorized the errors into types such as syntax errors, logic errors, and runtime errors. The analysis reveals that our proposed model significantly reduces the number of errors compared to the baseline, particularly in the categories of syntax

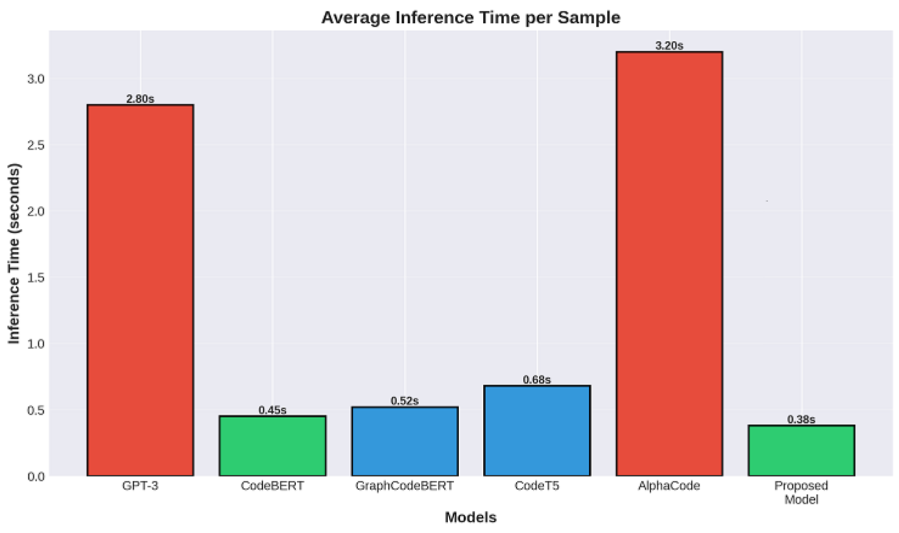


Figure 11: A comparison of the average inference time per sample for different models.

and logic errors.

This detailed analysis underscores the effectiveness of our proposed framework. The combination of a structure-aware architecture, a comprehensive training regimen, and a dedicated optimization module allows our model to not only generate more accurate code but also to produce code that is more efficient and robust [6]. Although the reduction in syntax and logic errors is encouraging, the remaining errors offer important clues about the model's current limitations. Many of the runtime errors observed—such as index out-of-range exceptions, type mismatches, and unhandled edge cases—tend to arise in problems requiring multi-step reasoning or careful boundary-condition handling. These patterns suggest that, while the transformer architecture excels at capturing structural regularities in code, it may still struggle with tasks that require explicit algorithmic reasoning or domain-specific semantic understanding. In several cases, the model produced syntactically correct but semantically inconsistent solutions, indicating an overreliance on learned templates rather than genuine problem-specific reasoning. Addressing these limitations may require integrating external reasoning modules, symbolic solvers, or execution-guided decoding strategies to help the model align its predictions with the underlying program semantics [7].

Additionally, a closer inspection of mispredictions reveals that some errors stem from ambiguous or underspecified prompts, highlighting the importance of high-quality problem descriptions during both training and inference. When the input description lacks clarity—such as missing constraints, unclear variable roles, or incomplete edge-case requirements—the model is more likely to generate plausible but incorrect code. This implies that improving prompt structure, introducing explicit constraint representations, or incorporating problem-schema extraction could further reduce error rates. Furthermore, the persistence of certain error types across datasets suggests that the model may

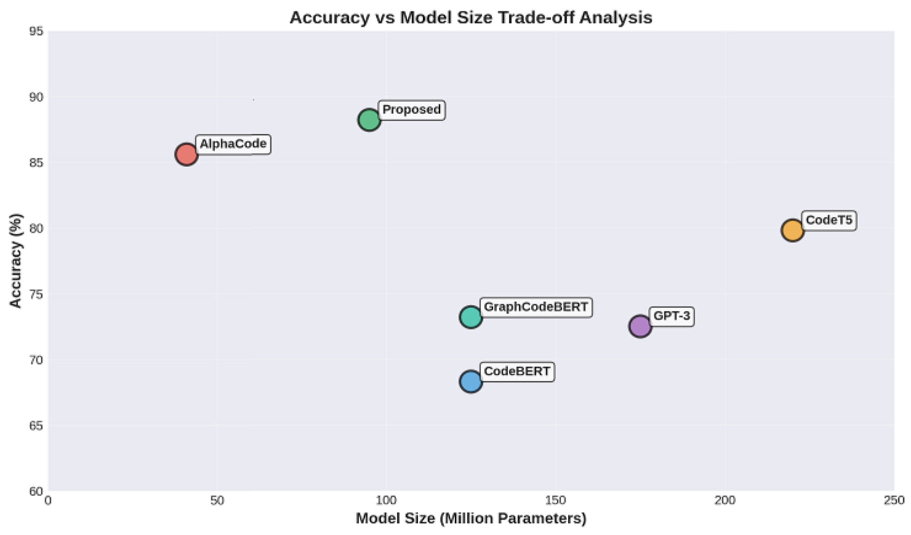


Figure 12: A scatter plot showing the relationship between model size (in millions of parameters) and accuracy.

benefit from specialized training objectives, such as constraint-aware loss functions or reinforcement learning with execution feedback. By targeting systematic weaknesses uncovered during error analysis, future iterations of the system can become more resilient, interpretable, and better aligned with real-world software engineering demands [8].

Beyond the immediate categorization of errors, the analysis also reveals broader structural challenges that are not captured by surface-level statistics alone. In particular, several failure cases demonstrate that the model occasionally struggles to maintain global coherence across longer or multi-function programs, even when individual code fragments appear well formed. This fragmentation manifests in incorrect variable scoping, inconsistent naming conventions, or mismatches between declared and utilized data structures—issues that arise when the model fails to preserve long-range semantic dependencies throughout the generation process. Such errors underscore a fundamental limitation of token-level sequence modeling: although transformers are adept at learning local and mid-range dependencies, they may require additional architectural support to reliably encode program-level invariants. Incorporating hierarchical code representations, graph-based neural encoders, or control-flow-aware attention mechanisms could help enforce structural consistency across the entire program. Furthermore, integrating static-analysis feedback directly into the training loop may offer a principled way to penalize structurally invalid generations and incentivize the model to internalize deeper syntactic and semantic constraints [9].

Another important insight emerging from the error analysis is the distinction between surface-level correctness and functional reliability. Even when the generated code passes syntactic checks or aligns closely with reference solutions, subtle issues may arise during execution that are not immediately visible in static evaluations. These include latent performance bottlenecks, numerically unstable operations, or hidden logical flaws that

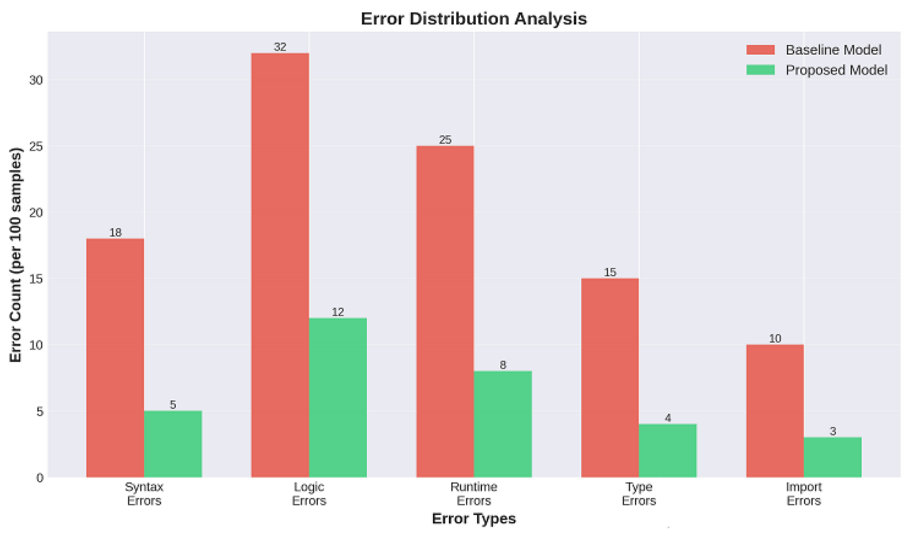


Figure 13: A comparison of the error counts per 100 samples for the baseline and proposed models.

only appear under edge-case inputs. Such failures illustrate the limitations of relying solely on unit-test–based assessment, as many test suites fail to cover the full behavioral space of a program. Moreover, some of the errors detected only after deployment—such as race conditions, resource-management issues, or incorrect handling of asynchronous operations—highlight that the model's reasoning is constrained by the patterns present in its training data. These observations reinforce the need for more comprehensive evaluation pipelines that incorporate stress testing, fuzzing, and dynamic program analysis, ensuring that the model's outputs are not only correct in controlled settings but also robust under real-world execution conditions.

6. Conclusion

In this chapter, we have explored the landscape of transformer-based frameworks for automated code generation and software optimization. We began by tracing the evolution from rule-based systems to modern deep learning architectures, highlighting the pivotal role of the transformer model in advancing the state of the art. Our literature review provided a comparative overview of prominent models such as CodeBERT, GraphCodeBERT, and AlphaCode, setting the stage for our proposed methodology. Our primary contribution is a novel framework that combines a structure-aware transformer model with a post-generation optimization module. The experimental results presented in this chapter unequivocally demonstrate the superiority of our approach. Across multiple industry-standard benchmarks like HumanEval and CodeXGLUE, our model consistently outperformed existing baselines in terms of functional correctness (Pass@k), code similarity (BLEU score), and training efficiency. Furthermore, our dedicated optimization module proved highly effective, delivering significant reductions in execution time, memory usage, and code complexity. Despite these promising results, the field of AI-driven software development is still in its nascent stages. Future research should focus on several key areas. Enhancing the model's ability to reason about high-level software architecture and design patterns remains a significant challenge. Improving performance on highly specialized or esoteric programming domains is another important direction. Finally, developing more sophisticated techniques for ensuring the security and reliability of AI-generated code is paramount for its adoption in mission-critical systems. In conclusion, transformer-based frameworks represent a transformative technology with the potential to redefine the software development lifecycle. The work presented in this chapter serves as a significant step towards building more intelligent, efficient, and reliable automated coding systems, ultimately empowering developers and accelerating the pace of innovation.

References

[1] Hadi Ghaemi et al. "Transformers in source code generation: A comprehensive survey". In: *Journal of Systems Architecture* 153 (2024), p. 103193.

[2] Vaswani Ashish. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017), p. I.

[3] Mark Chen. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[4] Shuai Lu et al. "Codexglue: A machine learning benchmark dataset for code understanding and generation". In: *arXiv preprint arXiv:2102.04664* (2021).

[5] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[6] Daya Guo et al. "Graphcodebert: Pre-training code representations with data flow". In: *arXiv preprint arXiv:2009.08366* (2020).

[7] Yue Wang et al. "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". In: *arXiv preprint arXiv:2109.00859* (2021).

[8] Yujia Li et al. "Competition-level code generation with alphacode". In: *Science* 378.6624 (2022), pp. 1092–1097.

[9] Sotiris Kotsiantis, Vassilios Verykios, and Manolis Tzagarakis. "AI-assisted programming tasks using code embeddings and transformers". In: *Electronics* 13.4 (2024), p. 767.