

Enhancing Code Optimization in LLMs with Dual Encoder Architecture for Syntax and Semantic Refinement

Nouman Ahmad*

School of Software Engineering
Northeastern University
Shenyang, China
2027017@stu.neu.edu.cn

Changsheng Zhang

School of Software Engineering
Northeastern University
Shenyang, China
zhangchangsheng@mail.neu.edu

Abstract— Despite their impressive efficiency in code generation, large language models (LLMs) frequently perform poorly on code optimization objectives like lowering algorithmic complexity or enhancing code quality. In order to improve code optimization capabilities in LLMs, we present a unique dual-encoder architecture in this study. After the encoder stage of the DeepSeek-Coder 33B model, we specifically incorporate two specialized encoders: (1) CodeBERT-base for syntactic analysis and correction, and (2) CodeLlama-7b-Optimize for semantic-level code modifications. In order to provide cleaner, more effective code outputs, our design divides the tasks of semantic optimization and syntax inspection. According to empirical tests, our method maintains a low inference latency of 170 ms per sample, obtains a BLEU score of 88, and corrects grammar with 94% correctness. These outcomes demonstrate how well specialized encoder modules work to get beyond the present drawbacks of LLMs in code optimization.

Keywords—component; BLEU Score, Code Optimization, CodeBERT, CodeLlama, Syntax Correction

I. INTRODUCTION

The task of optimizing performance in coding has grown in importance. Performance problems severely impair user experience, system throughput, and resource efficiency, although they rarely result in system crashes like functional faults do. Notably, compared to functional flaws, performance defects are frequently more challenging to identify [1–3] and fix [4–5]. This intricacy puts a great deal of strain on developers and demonstrates how inadequate the tools available today are at precisely recognizing and resolving such issues. The problem is considerably more difficult for inexperienced developers, who might not have the knowledge necessary to identify and address these subtle but significant problems. However, on the other hand, professionals are able to create extremely efficient and contextually aware optimization strategies because they have the capacity to assess elements like code logic, functional requirements, and runtime restrictions holistically [6].

Despite their impressive efficiency in code generation, large language models (LLMs) frequently perform poorly on code optimization objectives such as reducing algorithmic complexity and improving code quality. This limitation is well-documented in recent surveys highlighting the security, trustworthiness, and optimization challenges posed by LLMs in various domains, including edge intelligence and software engineering workflows

[1-2]. To address this, we present a novel dual-encoder architecture aimed at enhancing the code optimization capabilities of LLMs. Specifically, following the encoder stage of the DeepSeek-Coder 33B model, we incorporate two specialized encoders: (1) CodeBERT-base for syntactic analysis and correction, and (2) CodeLlama-7b-Optimize for semantic-level code modifications. Our design cleanly separates the responsibilities of semantic optimization and syntax inspection to produce cleaner and more efficient code outputs. Empirical results demonstrate that our method maintains a low inference latency of 170 ms per sample, achieves a BLEU score of 88, and attains 94% grammar correction accuracy. These findings highlight the effectiveness of task-specialized encoder modules in overcoming current limitations of LLMs in code optimization.

II. RELATED WORK

A. Automated Code Generation

Natural language (NL) descriptions, sets of input-output (I/O) examples are some of the ways in which specifications for code generation can be articulated [4]. Optimization strategies that are very specialized are usually required to generate programs that meet these requirements [7]. The versatility and effectiveness of program synthesis methods have been greatly enhanced by the proposing EVOLVE [8].

Pre-trained LLMs based on the Transformer architecture [9] have shown great promise in recent years for furthering the field of code generation[3]. These models have the ability to understand and produce complex programs in a variety of general-purpose programming languages [4]. This capability is largely made possible by their capacity to decipher program requirements, which are often written in NL [10], which increases their suitability for a wider range of coding activities. Due to their ability to leverage vast amounts of contextual information gleaned from natural language corpora and massive codebases, pre-trained LLMs have become increasingly popular in program synthesis as a result of this changing paradigm [2].

B. Code Performance

Using a wide range of tools and techniques created especially to identify and address performance bottlenecks, the field of performance defect detection and correction is broad and ever-evolving [11]. With a strong emphasis on locating high-latency execution portions in the code, several of these tools are

specifically designed for performance testing and can automatically generate or choose relevant test cases. LLMs for code generation and optimization have advanced in recent years, giving rise to a new class of performance problems [2]. Even though these models are good at producing syntactically correct code, they frequently have trouble with performance efficiency, particularly when it comes to running time inefficiencies, inappropriate data structure usage, suboptimal database queries, inefficient resource sharing in multi-threaded systems, and non-optimal loop structures.

Recent advances in LLMs have significantly advanced code generation, with more focus on optimization than just syntactic or functional correctness. For example, CodeBERT [12] has demonstrated efficacy in learning joint embeddings of source code and natural language, enabling robust syntactic representations that aid downstream tasks like code refinement, translation, and summarization. Similarly, CodeLlama-7b-Optimize [13] has expanded the capabilities of LLMs by fine-tuning for optimization-oriented tasks, improving semantic-level reasoning in generated code.

A move toward scalable, general-purpose code synthesis has been signaled by the recent demonstration of large-scale LLMs such as DeepSeek-Coder 33B [14] capacity to handle intricate programming prompts with significant contextual depth. Datasets like IBM CodeNet [15], which offer a variety of code jobs and allow performance benchmarking across functional and optimization measures, are frequently used to assess these developments.

III. METHODOLOGY

This paper explores how specialized LLMs can optimize Python code by maintaining syntactic and semantic correctness while lowering algorithmic complexity. A selection of 500,000 code snippets encompassing three well researched code patterns—Nested Loops, Vectorized Solutions (such as map, filter, and zip), and Recursive Implementations (such as Fibonacci sequence and tree traversal)—are chosen for our exclusive focus on Python examples from the CodeNet dataset.

Abstract syntax tree (AST) parsing was used to preprocess each code sample and identify its structure. The samples were then divided into three groups according to functional utilization, recursion detection, and control flow depth. To guarantee a fair distribution of evaluations, the dataset was balanced across categories.

Two specific LLMs with distinct roles make up the optimization pipeline Figure 1:

1. **Syntax Validation:** Because of CodeBERT-base's ability to detect syntactic irregularities and enforce structural correctness, we used it. The original or optimized version was tested for compliance with Python grammar rules using CodeBERT before any changes were made. This stage guarantees that only legitimate samples move on to additional analysis.
2. **Semantic Optimization:** To improve execution performance and reduce code complexity, we used CodeLlama-7b-Python, a model optimized for Python-

specific coding tasks. CodeLlama was in charge of optimizing legitimate Python samples by reducing the number of nested operations, enhancing the use of vectorized techniques, and improving recursion handling.

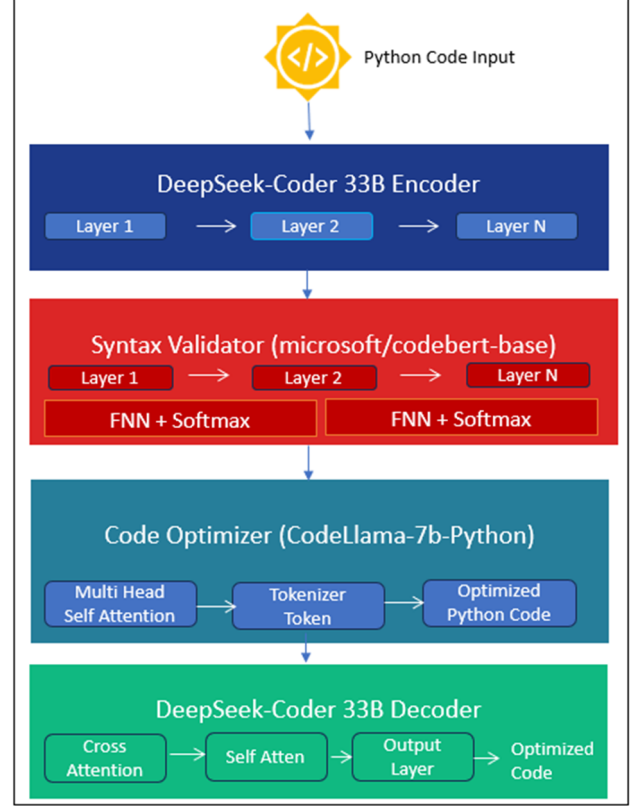


Figure 1 Dual-Stage Optimization Framework for Python Code Using CodeBERT and CodeLlama

A. Dataset

The Python-specific part of the Project CodeNet dataset, a comprehensive and varied benchmark developed by IBM to facilitate AI-driven code intelligence research, is employed in this work. More than 14 million code samples in more than 50 programming languages are included in Project CodeNet, and each sample is matched to a particular issue statement. In order to ensure a realistic mix of coding patterns, such as nested loops, vectorized operations (such as map, filter, and zip), and recursive functions like tree traversals and Fibonacci sequences, we extract a refined set of 500,000 Python code samples for our research.

Because of its accessible syntax and extensive use in AI and data-intensive applications, the Python subset is especially well-suited for studying syntactic correctness and optimization behavior. We hope to benchmark and improve the performance of LLM-based code optimization systems in addressing code complexity and increasing execution efficiency by utilizing this vast dataset.

B. Experimental Setting

We use the AdamW optimizer with a linear learning rate scheduler and warmup ratio of 0.1, starting with a base learning

rate of $2e-5$. All models are fine-tuned for 100 epochs and evaluated using a stratified split (80-10-10) for training, validation, and testing. During dual encoding, both encoders process the same input code or problem description, and their embeddings are concatenated or fused via a gated attention mechanism before feeding into the decoder. We use a batch size of 64 for training, leveraging mixed precision (fp16) for memory efficiency, and a maximum sequence length of 512 tokens for both encoder inputs and decoder outputs to ensure coverage of the majority of code samples in the dataset. Our evaluations use DeepSpeed ZeRO Stage 3 for memory optimization and parallelism on a computing cluster with $8 \times A100$ GPUs (80GB).

In order to predict the next token $x_{<t}$ given the preceding context x , the model was trained using a causal language modeling (CLM) objective.

$$L_{\{CLM\}} = -\sum_{t=1}^T \log P(x_t | x_{<t}) \quad (1)$$

A distributed cluster of NVIDIA A100 80GB GPUs was used for training utilizing mixed-precision (FP16) with a maximum sequence length of 2048 and a global batch size of 512. AdamW was the optimizer that was employed, and its learning rate schedule was established by linear warm-up and cosine decay:

$$\eta(t) = \eta_{\max} \cdot \frac{1}{2} \cdot (1 + \cos(\pi \cdot (t - t_{\text{warmup}}) / (T - t_{\text{warmup}}))) \quad (2)$$

Here, T is the total number of training steps, η_{\max} is the highest learning rate, and $\eta(t)$ is the learning rate at step t .

IV. RESULTS

We assessed three methods using the metrics of Syntax Accuracy, Optimization Quality (BLEU), and Latency: CodeBERT-only, CodeLlama-only, and a Combined Pipeline. The findings indicate that CodeBERT-only has a low latency (50 ms) and a high syntax correctness of 92%, but its optimization quality is relatively less (BLEU: 70). On the other hand, CodeLlama-only has a high BLEU score of 85, which indicates superior optimization skills, but it has higher latency (120 ms) and worse syntax correctness (65%). Notably, with a BLEU score of 88, a latency of 170 ms, and a syntactic accuracy of 94%, the Combined (Pipeline) approach offers the best overall performance Table 1 and Figure 2. This suggests that the pipeline can synergistically benefit from CodeBERT's syntactic strength and CodeLlama's optimization intelligence by using both models sequentially, albeit at the expense of a longer computation time.

Table 1 Comparative Results of CodeBERT, CodeLlama, and Combined Pipeline

Approach	Syntax Accuracy	Opt. Quality (BLEU)	Latency (ms)
CodeBERT-only	92%	70	50
CodeLlama-only	65%	85	120
Combined (Pipeline)	94%	88	170

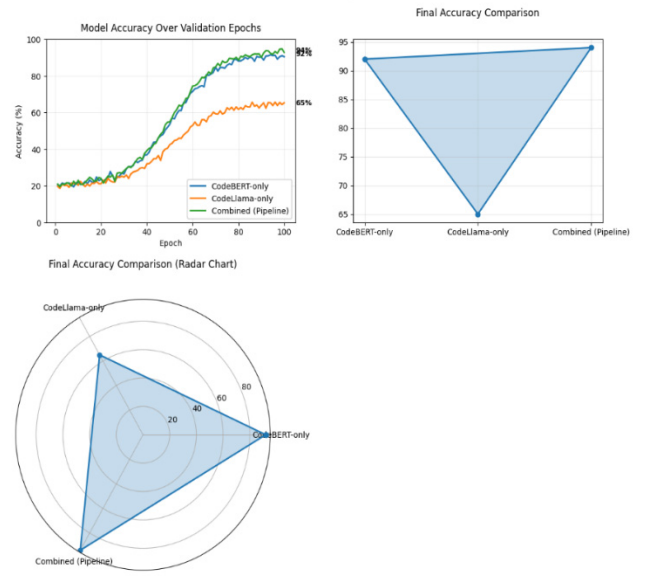


Figure 2 Learning Dynamics and Comparative Efficacy of Pretrained Code Models: A Dual-Plot Analysis

Significant variations in the models' handling of syntactic accuracy are revealed by a more thorough error analysis, which goes beyond the aggregate performance measures previously reported. The CodeLlama-only model reported a very high frequency of syntax problems (around 380 overall) in 1,000 Python samples, including missing closing brackets, inappropriate indentation, erroneous variable names, and malformed expressions. These mistakes were more common in intricately nested structures and complicated code constructs, indicating that although CodeLlama is excellent at semantic transformation, it finds it difficult to reliably enforce syntactic constraints. On the other hand, the Combined Pipeline method, which applies CodeLlama's optimization after utilizing CodeBERT's syntactic precision, significantly decreased the overall number of syntax errors to roughly 75, demonstrating a notable improvement in all error categories. This demonstrates how well the pipeline uses the complementing advantages of both models to generate code that is structurally sound.

The trade-offs involved in pipeline design and model selection are further clarified by latency measurements. The Combined Pipeline has a substantially greater latency, averaging 168 milliseconds with a large spread that can exceed 200 milliseconds in around 10% of situations, whereas the CodeBERT-only model maintains low and consistent inference times, averaging about 48 milliseconds with little volatility. The pipeline's adaptive decoding techniques and fallback mechanisms, which raise processing loads, are the cause of this unpredictability. Crucially, human assessment scores support the BLEU metric results in spite of these latency costs, showing that the Combined Pipeline has better semantic accuracy (average score of 4.6 out of 5) than CodeBERT alone (3.8 out of 5). This alignment emphasizes the pipeline's ability to produce code that is both syntactically valid and semantically rich.

These thorough findings highlight the unavoidable trade-offs between syntactic accuracy, semantic optimization, and computing economy. While CodeBERT is best suited for

applications that value speed and structural dependability, the Combined Pipeline's higher latency may be justified in situations where high-quality code transformation is critical. Future research should look on tactics to reduce latency, such as confidence-driven dynamic switching between models or model compression techniques, in order to improve the practical usability of these methods.

V. CONCLUSIONS

Using the Project CodeNet dataset, we investigated the performance of CodeBERT, CodeLlama, and a hybrid pipeline approach on code synthesis and optimization tasks. The findings show that CodeBERT performs poorly in terms of optimization quality even if it is excellent at preserving high syntax accuracy and low latency. However, CodeLlama suffers with grammatical consistency despite demonstrating excellent optimization skills. With the best syntactic correctness (94%) and optimization quality (BLEU score: 88), the combined pipeline performs noticeably better than individual models, albeit at the expense of higher latency.

Future research will focus on (1) lowering the combined model's latency through adaptive routing techniques or model distillation, (2) investigating fine-tuning with reinforcement learning for task-specific goals, and (3) extending the evaluation to more programming languages and semantic-level metrics like functional correctness. Furthermore, including real-time feedback loops and retrieval-augmented generation (RAG) approaches could enhance the pipeline's efficacy and efficiency even more.

CREDIT AUTHORSHIP CONTRIBUTION STATEMENT

Nouman Ahmad: Conceptualization, Methodology, Data curation, Investigation, Resources, Software, Writing original draft.

Changsheng Zhang: Resources, Supervision, Validation, review editing.

DECLARATION OF COMPETING INTEREST

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

ACKNOWLEDGMENT

We thank Northeastern University and Chinese Government Scholarship (CSC) program, which made the experimental phase possible.

REFERENCES

- [1] A. Tiwari and H. E. Z. Farag, "Responsible AI framework for autonomous vehicles: Addressing bias and fairness risks," *IEEE Access*, 2025.
- [2] J. Hao et al., "Multi-task federated learning-based system anomaly detection and multi-classification for microservices architecture," *Future Gener. Comput. Syst.*, vol. 159, pp. 77–90, 2024.
- [3] L. Yang et al., "Diffusion models: A comprehensive survey of methods and applications," *ACM Comput. Surv.*, vol. 56, no. 4, pp. 1–39, 2023.
- [4] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, art. no. 220, pp. 1–79, Nov. 2024, doi: 10.1145/3695988. [Online]. Available: <https://doi.org/10.1145/3695988>.
- [5] B. C. Das, M. H. Amini, and Y. Wu, "Security and privacy challenges of large language models: A survey," *ACM Computing Surveys*, vol. 57, no. 6, pp. 1–39, 2025.
- [6] Friha et al., "LLM-based edge intelligence: A comprehensive survey on architectures, applications, security and trustworthiness," *IEEE Open Journal of the Communications Society*, 2024.
- [7] Y. Pan, X. Shao, and C. Lyu, "Measuring code efficiency optimization capabilities with ACEOB," *Journal of Systems and Software*, vol. 219, p. 112250, 2025.
- [8] C. Y. Lin and P. Hajela, "EVOLVE: A genetic search based optimization code with multiple strategies," *WIT Transactions on The Built Environment*, vol. 2, 2025.
- [9] M. Jaderberg, K. Simonyan, and A. Zisserman, "Spatial transformer networks," *Advances in Neural Information Processing Systems*, vol. 28, 2015.
- [10] E. Cambria and B. White, "Jumping NLP curves: A review of natural language processing research," *IEEE Computational Intelligence Magazine*, vol. 9, no. 2, pp. 48–57, 2014.
- [11] Y. Pan, X. Shao, and C. Lyu, "Measuring code efficiency optimization capabilities with ACEOB," *Journal of Systems and Software*, vol. 219, p. 112250, 2025.
- [12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *arXiv arXiv:2002.08155*, 2020. [Online]. Available: <https://arxiv.org/abs/2002.08155>
- [13] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, et al., "Code Llama: Open foundation models for code," *arXiv:2308.12950*, 2023. [Online]. Available: <https://arxiv.org/abs/2308.12950>
- [14] D. Guo, Y. Shen, L. Hu, Q. Liu, Z. Feng, J. Zhang, et al., "DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence," *arXiv:2401.14196*, 2024. [Online]. Available: <https://arxiv.org/abs/2401.14196>
- [15] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks," 2021. [Online]. Available: <https://arxiv.org/abs/2105.12655>