The Journal of Supercomputing (2025) 81:1006
https://doi.org/10.1007/s11227-025-07378-5

Efficient program optimization through knowledge-enhanced LoRA fine-tuning of large language models

Caixu Xu[1] · Hui Guo[1,2] · Caicun Cen[1,3] · Minglang Chen[1,3] · Xiongjie Tao[2] · Jie He[1]

Accepted: 1 May 2025 / Published online: 10 June 2025

Abstract
Source code optimization enables developers to enhance programs at the human–computer interaction level, thereby improving development efficiency and product quality. With the rise of large language models (LLMs), fine-tuning and prompting have become mainstream solutions for this task. However, both approaches present challenges: fine-tuning is resource-intensive due to the exponential growth in the scale of LLMs, whereas prompting, although resource-efficient, struggles to generate high-quality optimized programs. In this paper, we present CodeOPT, a LoRA-driven approach for fine-tuning LLMs to optimize C/C++ code. Instead of fine-tuning all LLM parameters, CodeOPT leverages LoRA to fine-tune only an optimization adapter, significantly reducing the number of trainable parameters. Additionally, we incorporate prior optimization knowledge during fine-tuning and introduce optimization-based instruction fine-tuning, enabling LLMs to effectively learn from external knowledge sources to improve program optimization. To evaluate the effectiveness of CodeOPT, we benchmarked it against several baselines on challenging programming tasks from different code completion platforms. Experimental results demonstrate that CodeOPT outperforms all baselines, including the state of the art, while keeping modifications to the original program minimal.

Keywords Source code optimization · Parameter-efficient training · Instruction fine-tuning

1 Introduction

Software optimization is a process of refining a program to better utilize resources, e.g., less time, memory, and energy while maintaining the original functionality. In previous decades, this process has been achieved at the human level or compiler level. Specifically, developers utilize expertise and domain knowledge to improve

Extended author information available on the last page of the article

the program by taking more efficient data structures or algorithms at the source code level. As for the compiler level, it generally utilizes different automated optimization methods for the intermediate representation to enhance the performance of the program.

Optimizing software at the source code level offers significant performance benefits by allowing developers to incorporate domain-specific knowledge that automated systems often lack. This approach enables fine-grained, context-aware optimizations that address both general performance considerations and application-specific requirements. For instance, developers can select specialized algorithms or data structures better suited to the problem at hand. Moreover, source code-level modifications are more interpretable for developers, facilitating easier maintenance and future adaptations compared to automated optimization techniques for compilers.

SUPERSONIC [1], an initial approach for source code optimization, marks a notable advancement in applying machine learning to software optimization. It builds upon previous research by integrating large language models (LLMs) to identify optimization opportunities in source code. Unlike conventional approaches that rely on predefined rules for automated transformations, SUPERSONIC utilizes the flexibility of LLMs within a Seq2Seq framework, enabling end-to-end code optimization. Specifically, given an input program, it can generate diff-based outputs to suggest potential improvements. Experimental results indicate that SUPERSONIC serves as a promising state-of-the-art (SOTA) solution, effectively bridging the gap between domain-specific knowledge and data-driven optimization techniques, with performance surpassing that of ChatGPT-3.5-turbo and GPT-4.

While SUPERSONIC offers valuable capabilities, it does have certain limitations that affect its applicability and efficiency. First, its reliance on full-parameter fine-tuning requires substantial GPU resources, which can be cost-prohibitive and restrict scalability. Additionally, the diff-based output format used by SUPERSONIC is rarely encountered by large language models during pre-training, which introduces challenges during the fine-tuning process. Second, the model currently takes only the program in need of optimization as input, without considering general C/C++ optimization guidelines. This type of input format may lead to suboptimal results, as the model lacks explicit awareness of the common optimization strategies typically used by developers. Incorporating such domain-specific knowledge could enhance the model's effectiveness in recognizing and applying appropriate optimizations.

To address the issues identified in SUPERSONIC, we propose CodeOPT, a LoRA-driven [2], LLM-based [3, 4] approach for optimizing C/C++ source code. Rather than performing full-parameter fine-tuning, we opt to fine-tune an optimization adapter, allowing us to adjust only a small subset of the trainable parameters (approximately $\frac{1}{1750}$ of the LLM). To maintain consistency with pre-training, we employ decoder-only LLMs as the foundation model, fine-tuning them to generate optimized code in an autoregressive manner. Additionally, to effectively guide the LLMs in learning program optimization, we extend traditional fine-tuning to instruction-based fine-tuning. Specifically, we summarize standard C/C++ optimization strategies and provide them as additional inputs, aiding the LLMs in learning how to optimize C/C++ programs and produce high-quality, optimized code.

To assess the effectiveness of CodeOPT, we conducted all experiments on an open-source dataset provided by Chen et al. [1], comparing our results against state-of-the-art baseline approaches, including SUPERSONIC, ChatGPT-3.5-turbo, and GPT-4. We select ChatGPT family since they have been proven the state-of-the-art performance in various software engineering tasks [5–13]. Specifically, we chose LLaMA 3.2 as the base model and fine-tuned it using instruction-based methods on the training set, saving the checkpoint with the best performance on the evaluation set. We then evaluated the final performance of CodeOPT on the testing set. The experimental results demonstrate that CodeOPT was able to optimize more programs within the testing set. Furthermore, when considering average time and memory reduction of the optimized programs, CodeOPT achieved performance comparable to ChatGPT-3.5-turbo and outperformed SUPERSONIC by up to 26.79% (3.36× vs. 2.65×) and 244.20% (6.23× vs. 1.81×), highlighting the effectiveness of CodeOPT.

In this paper, we make the following contributions,

- We introduce CodeOPT, a novel LLM-based approach for optimizing C/C++ source code. To enhance the quality of optimization, we utilize decoder-only LLMs as the foundation and fine-tune it to generate optimized programs in an autoregressive manner, maintaining consistency between pre-training and fine-tuning. Additionally, we extend conventional fine-tuning by incorporating instruction fine-tuning, which involves adding standard optimization strategies to the input. This approach effectively guides the model in learning improved optimization techniques.
- To reduce computational resource costs, we propose fine-tuning an optimization adapter, which enables us to adjust only a small subset of trainable parameters—approximately $\frac{1}{1750}$ of the parameters of the original LLMs.
- To assess the effectiveness and efficiency of CodeOPT, we conducted a comparative analysis against state-of-the-art models, including ChatGPT-3.5-turbo and GPT-4. The results demonstrate the superior optimization capabilities of our approach.

The remainder of this paper is organized as follows. Section 2 introduces the background knowledge, and Sect. 3 elaborates our proposed approach, CodeOPT. Section 4 and Sect. 5 present the experimental setups and results, respectively. Section 6 discusses the threats to validity. Finally, we conclude this paper in Sect. 7.

2 Background

In this section, we introduce the background knowledge for our work, including code optimization, large language models (LLMs), parameter-efficient fine-tuning, and related work.

2.1 Code optimization

Source code optimization [1, 14] refers to the process of transforming a program's source code to improve its performance without altering its intended behavior. Given an input, which is the original version of a source code, the goal of optimization is to produce an output-a transformed version of the code—that is functionally equivalent but exhibits improvements such as reduced runtime, lower memory usage, or other efficiency metrics. To ensure that the output is indeed an optimized version, the optimized code must be thoroughly tested to verify functional correctness and assessed against specific metrics like execution speed or resource usage. The optimization process involves various techniques, such as loop unrolling, code simplification, and data structure selection, aimed at minimizing computational overhead while preserving program correctness.

To illustrate the concept of source code optimization, consider the example of a bubble sort algorithm. In the original version, two nested `for` loops are used to iterate over the array and repeatedly swap adjacent elements until the array is sorted. In the optimized version, the `for` loops have been refined to reduce unnecessary comparisons, resulting in a significant reduction in runtime complexity for cases where the array becomes sorted before all iterations are complete. This optimization yields improved efficiency while maintaining the sorting functionality of the original implementation. Figure 1 provides a comparison of the original and optimized versions of bubble sort, highlighting the differences in loop structure.

Listing 1 Original Bubble Sort

```
// Original Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1;
             j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and
                    arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Listing 2 Optimized Bubble Sort

```
// Optimized Bubble Sort
void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = false;
        for (int j = 0; j < n-i-1;
             j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and
                    arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = true;
            }
        }
        // If no elements were
            swapped, array is
            sorted
        if (swapped == false)
            break;
    }
}
```

Fig. 1 Comparison of original and optimized bubble sort algorithms

2.2 Large language model

Large language models [3, 15] are typically categorized into three main types: encoder-only [3], decoder-only [15], and encoder-decoder models [16]. Encoder-only models, such as BERT [3], are well-suited for understanding tasks, while decoder-only models, such as GPT [15], are designed for generative tasks. Encoder-decoder models, like T5 [16], combine the strengths of both architectures, making them versatile for a wide range of natural language processing tasks. ChatGPT [15], based on the decoder-only architecture, exemplifies the power of generative LLMs. One key advantage of decoder-only LLMs is their ability to maintain consistency between pre-training and fine-tuning, enabling efficient transfer learning for various downstream tasks. This feature makes models like ChatGPT particularly effective in adapting to new tasks by multiple methods, including zero-shot learning, few-shot learning, and fine-tuning.

LLMs have shown great promise in software engineering tasks. They have been successfully applied to various activities, such as code search [5, 6], bug report analysis [8, 17], code generation [18], test data generation [13], program repair [12, 19], generating pull request titles [7, 20], and vulnerability detect for smart contract [21–23]. These capabilities can significantly enhance the productivity of software developers by automating labor-intensive tasks and providing intelligent suggestions that improve code quality and development efficiency.

2.3 Parameter-efficient fine-tuning

With the increasing size of large language models [4], the number of trainable parameters has grown exponentially, which requires significant computational resources and GPU memory to fine-tune. Parameter-efficient fine-tuning methods have emerged as a solution to address these challenges by enabling the adaptation of pre-trained models with fewer trainable parameters. These approaches are crucial for making fine-tuning feasible and accessible for practical usage. One prominent parameter-efficient fine-tuning technique is low-rank adaptation (LoRA) [2]. LoRA introduces trainable low-rank matrices that are added to the frozen pre-trained model weights, effectively reducing the number of parameters that need to be updated during fine-tuning. By only optimizing these low-rank matrices, LoRA can significantly reduce the computational and memory requirements compared to full model fine-tuning while still maintaining competitive performance.

Several studies have applied LoRA to various software engineering tasks. For instance, Wang et al. [24] fine-tuned a single adapter for both code search and code summarization tasks using a dataset spanning multiple programming languages, demonstrating the effectiveness of LoRA-based parameter-efficient fine-tuning. Moreover, Silva et al. introduced RepairLLaMA [12], fine-tuning a repair adapter specifically for fixing Java bugs. Their results indicate that parameter-efficient fine-tuning can outperform full-parameter fine-tuning. Hence, in this work, we design an optimization adapter for optimizing C/C++ source code.

2.4 Related work

2.4.1 Deep learning for code optimization

Recent studies increasingly focus on applying deep learning to optimize source code. For instance, SUPERSONIC [1] fine-tunes a C++ version of CodeBERT using a Seq2Seq framework to generate diff-based output in an end-to-end manner. Experimental results show that SUPERSONIC optimizes more C/C++ programs compared to ChatGPT-3.5-turbo and GPT-4. Another example, PIE4Perf [14], explores large language models for performance-improving code edits, evaluating multiple models using different approaches such as fine-tuning and few-shot learning. PIE-4Perf achieves performance improvements in over 25% of programs, with speedups exceeding 2.5x in many cases. DeepDev-PERF [25], another related approach, targets optimizing C# programs by pre-training a language model on code and natural language corpora, followed by fine-tuning for performance enhancement. A novel aspect of DeepDev-PERF is its use of developer benchmarks to evaluate effectiveness, ensuring reliable results. Other related works include tools like Artemis++, which employs genetic algorithms to optimize data structures, and RAPGen [26], which uses Codex for addressing C# inefficiencies through zero-shot prompting.

Building upon our review of existing approaches, our CodeOPT methodology introduces several key advancements that distinguish it from previous work. Unlike SUPERSONIC, which uses encoder-only models with full-parameter fine-tuning, we leverage decoder-only LLMs with parameter-efficient LoRA fine-tuning, reducing trainable parameters to approximately 1/1750 of the original model while maintaining consistency between pre-training and fine-tuning phases. In contrast to PIE4Perf and DeepDev-PERF, we incorporate domain-specific optimization knowledge directly into the training process through instruction-based fine-tuning, enabling the model to learn from established C/C++ optimization strategies rather than relying solely on examples in the training data. Furthermore, while RAPGen employs zero-shot prompting that depends entirely on pre-trained knowledge, our approach combines the strengths of both pre-trained foundations and targeted fine-tuning, resulting in a model that can generate more effective and diverse optimizations across a broader range of programs as demonstrated by our experimental results.

2.4.2 LLMs for software engineering

LLMs have also been extensively utilized in various software engineering tasks, spanning from automated program repair to code generation and refactoring. RepairLLaMA [12] introduces a fine-tuning pipeline for automated program repair, leveraging parameter-efficient techniques like LoRA to adapt LLMs effectively for the repair task. By incorporating realistic fault localization signals, RepairLLaMA significantly enhances the performance of program repair compared to naive code representations. Recent LLMs tailored for software engineering, such as CodeBERT [11] and CodeT5 [27], have been fine-tuned for tasks like code summarization [10], code search [5, 6], and program repair [12], showing notable improvements in understanding and generation capabilities across multiple programming languages.

These advancements reflect the growing trend of using LLMs to support and automate various phases of software development, ultimately improving code quality and productivity.

3 Approach

In this section, we introduce the pipeline of CodeOPT, including parameter-efficient fine-tuning, instruction-based training, and inference (Fig. 2).

3.1 Model introduction

Due to the significant performance of recent decoder-only large LLMs [15, 28, 29], we select LLaMA 3 with 7B size [18] as the foundational model for this work. LLaMA 3 [30] is Meta's newest advancement in foundational AI models. It is engineered to excel in multilingual tasks [31, 32], reasoning [30], coding [33, 34], and even tool usage [35]. The model boasts a dense Transformer architecture [36] with up to 405 billion parameters and supports a context window of up to 128,000 tokens. For its pre-training, LLaMA 3 was fed an enormous dataset of about 15 trillion multilingual tokens, significantly surpassing the data size of its predecessors [15,

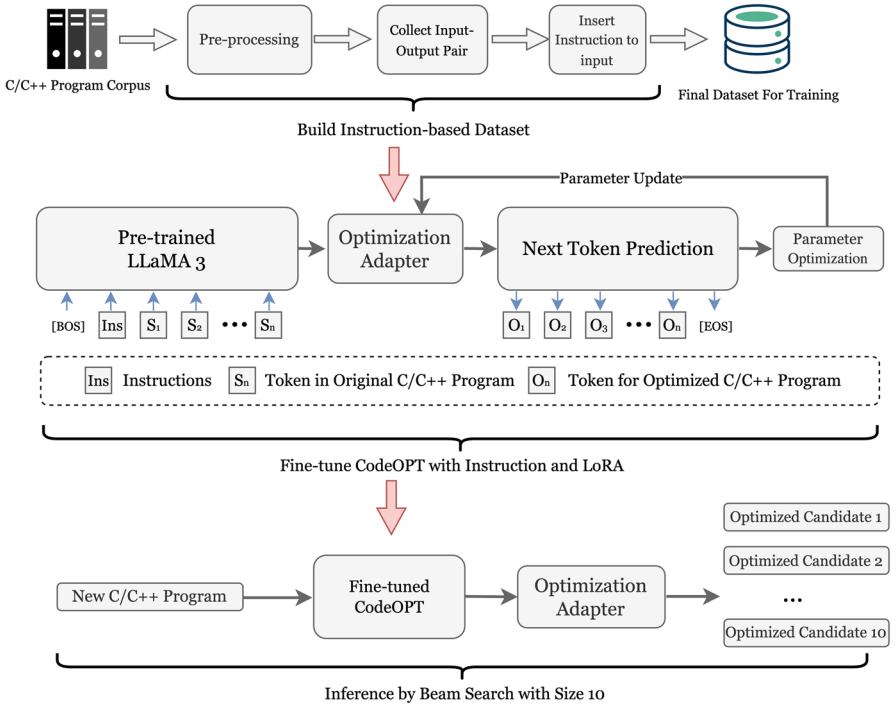

Fig. 2 The pipeline of CodeOPT

18]. The training process involved two main phases: an initial pre-training focused on next-token prediction, followed by a post-training phase that included supervised fine-tuning and direct preference optimization. Specifically, LLaMA 3 integrates capabilities for processing images, videos, and speech. Empirical evaluations indicate that it performs on par with leading models like GPT-4 across a variety of tasks. Even its smaller versions outperform other models in their size category. More importantly, LLaMA 3 provides a better balance between task performance and safety. It incorporates the LLaMA Guard 3 system[1] to mitigate risks associated with harmful outputs, ensuring more responsible AI interactions.

3.2 Parameter-efficient fine-tuning with LoRA

Large language models (LLMs) like LLaMA 3 have demonstrated remarkable capabilities across various natural language processing tasks. However, fine-tuning these models for specific applications can be computationally intensive due to their vast number of parameters. Low-rank adaptation (LoRA) offers a parameter-efficient alternative by introducing trainable low-rank matrices into each layer of the model, significantly reducing the number of parameters that need to be updated during fine-tuning.

In the transformer architecture underlying LLaMA 3, the core computations involve weight matrices in the self-attention and feed-forward layers. Consider a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ in any linear transformation within the model. Instead of updating $\mathbf{W}$ directly during fine-tuning, LoRA keeps $\mathbf{W}$ frozen and introduces a low-rank decomposition:

$$\Delta \mathbf{W} = \mathbf{B}\mathbf{A}, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$ and $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$ are the trainable low-rank matrices, and $r \ll \min(d_{\text{in}}, d_{\text{out}})$ is the rank. The modified weight matrix becomes:

$$\mathbf{W}' = \mathbf{W} + \Delta \mathbf{W}. \tag{2}$$

During the forward pass, the transformation is computed as:

$$\mathbf{h}' = \mathbf{W}'\mathbf{x} = \mathbf{W}\mathbf{x} + \mathbf{B}(\mathbf{A}\mathbf{x}), \tag{3}$$

where $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ is the input and $\mathbf{h}' \in \mathbb{R}^{d_{\text{out}}}$ is the output. By only training $\mathbf{A}$ and $\mathbf{B}$, LoRA reduces the number of trainable parameters from $d_{\text{in}} \times d_{\text{out}}$ to $r \times (d_{\text{in}} + d_{\text{out}})$, which is significantly smaller when r is small.

Fine-tuning the LLaMA 3 model with LoRA involves updating only the low-rank matrices $\mathbf{A}$ and $\mathbf{B}$ while keeping the original weights $\mathbf{W}$ fixed. The model is trained to predict the next token in a sequence, which is formulated as a language modeling task. Given a sequence of tokens $t_1, t_2, \ldots, t_n$, the objective is to maximize the likelihood:

[1] https://huggingface.co/meta-llama/Llama-Guard-3-8B.

$$\mathcal{L} = \sum_{i=1}^{n} \log P(t_i | t_{<i}; \theta, \Delta\theta), \tag{4}$$

where θ represents the original model parameters (kept fixed), and $\Delta\theta$ represents the parameters of the low-rank matrices introduced by LoRA. The probability of the next token is computed using the softmax function over the output logits:

$$P(t_i | t_{<i}; \theta, \Delta\theta) = \text{softmax}(\mathbf{h}_{i-1}^{\top} \mathbf{E}), \tag{5}$$

where $\mathbf{h}_{i-1}$ is the hidden state obtained from the modified weights $\mathbf{W}'$, and $\mathbf{E}$ is the token embedding matrix. The gradients with respect to $\mathbf{A}$ and $\mathbf{B}$ are computed using backpropagation:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{A}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}'} \frac{\partial \mathbf{h}'}{\partial \mathbf{B}}. \tag{6}$$

These gradients are then used to update $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{A} \leftarrow \mathbf{A} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{A}}, \quad \mathbf{B} \leftarrow \mathbf{B} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{B}}, \tag{7}$$

where η is the learning rate.

Now, we introduce how to fine-tune with LoRA for source code optimization. The goal is to train the model to map an input code snippet **c**in to an optimized output code snippet **c**out. The fine-tuning process aims to minimize the difference between the model's output and the ground truth optimized code. The objective function can be formulated as:

$$\mathcal{L} = \sum_{i=1}^{N} \mathcal{L}\text{seq}(\mathbf{c}\text{out}^{(i)}, f_{\theta+\Delta\theta}(\mathbf{c}_{\text{in}}^{(i)})), \tag{8}$$

where $\mathcal{L}_{\text{seq}}$ is the sequence-to-sequence loss function (e.g., cross-entropy loss), N is the number of training examples, θ represents the original model parameters (kept fixed), and $\Delta\theta$ represents the LoRA parameters. During fine-tuning, the model processes the input code using the modified weights $\mathbf{W}'$:

$$\mathbf{h}' = f_{\theta+\Delta\theta}(\mathbf{c}_{\text{in}}), \tag{9}$$

where $f_{\theta+\Delta\theta}$ denotes the LLaMA model augmented with LoRA. The output is then generated by decoding $\mathbf{h}'$ to produce the optimized code snippet.

Considering the unoptimized C++ code snippet in Fig. 3, a common optimization is to implement memoization, which stores previously computed values to avoid redundant calculations, as shown in Fig. 4.

The fine-tuning process trains the model to perform such transformations. The input token sequence $\mathbf{c}_{\text{in}}$ is the unoptimized code tokenized appropriately, and the target sequence $\mathbf{c}_{\text{out}}$ is the optimized code. When using LoRA, we update only the low-rank adaptation matrices $\mathbf{A}$ and $\mathbf{B}$ while keeping the original model weights θ fixed.

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

void calculateFibonacci(int count) {
    for (int i = 0; i < count; i++) {
        std::cout << "Fibonacci(" << i << ") = "
                  << fibonacci(i) << std::endl;
    }
}
```

Fig. 3 Unoptimized recursive Fibonacci implementation

```
int fibonacci(int n, std::vector<int>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];

    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo);
    return memo[n];
}

void calculateFibonacci(int count) {
    std::vector<int> memo(count, -1);
    for (int i = 0; i < count; i++) {
        std::cout << "Fibonacci(" << i << ") = "
                  << fibonacci(i, memo) << std::endl;
    }
}
```

Fig. 4 Optimized Fibonacci implementation using memoization

By incorporating LoRA into the fine-tuning process for source code optimization, we can obtain the following improvements:

- **Efficiency**: Only a small number of parameters are updated, reducing computational resources.
- **Scalability**: Enables fine-tuning on large datasets of code without extensive hardware.

3.3 Instruction-based fine-tuning

Instruction-based fine-tuning has emerged as a pivotal technique in enhancing the performance of large language models (LLMs) for specific tasks. By fine-tuning models with task-specific instructions, we can guide them to produce more accurate and relevant outputs. In the context of source code optimization, leveraging instructions that encapsulate standard C/C++ optimization methods can guide the model effectively in learning how to optimize.

Instruction fine-tuning involves training an LLM on a dataset composed of instruction-output pairs. The model learns to generate outputs that are coherent with the given instructions, effectively aligning its generation process with desired

behaviors. Formally, let $\mathcal{D} = (I_n, O_n)_{n=1}^{N}$ be a dataset where I_n is an instruction and O_n is the corresponding desired output. The objective is to minimize the loss function:

$$\mathcal{L} = -\sum_{n=1}^{N}\sum_{t=1}^{T_n} \log P\left(o_t^{(n)} \mid I_n, o_{<t}^{(n)}; \theta\right), \tag{10}$$

where $o_t^{(n)}$ is the t-th token of the output O_n, $o_{<t}^{(n)}$ denotes all previous tokens before t, and θ represents the model parameters.

To tailor the LLM for source code optimization, we introduce instructions based on standard C/C++ optimization techniques. After analyzing common optimization patterns in our dataset and considering techniques that offer substantial performance improvements while maintaining code correctness, we selected four fundamental optimization strategies:

- **Loop unrolling**: Transform loops to reduce the overhead of loop control code.
- **Inline expansion**: Replace function calls with the function body to eliminate call overhead.
- **Constant folding**: Compute constant expressions at compile time to reduce runtime computations.
- **Dead code elimination**: Remove code that does not affect the program results to optimize performance.

These four techniques were deliberately chosen based on several criteria: (1) they represent core strategies widely recognized in compiler optimization literature, (2) they address different aspects of program execution (control flow, function calls, compile time computation, and code redundancy), (3) they are applicable across diverse C/C++ programs regardless of application domain, and (4) they have proven effectiveness in improving both execution time and memory usage. While numerous other optimization techniques exist (such as loop fusion, strength reduction, or memory alignment), these four provide an optimal balance between coverage of common optimization opportunities and specificity in guiding the model toward meaningful transformations.

By integrating the above optimization instructions, we finally can get the following instruction:

You are a helpful coding assistant and your task is to optimize the given C/C++ program.

I will give you some basic optimization instructions, and you can optimize the given program according to them.

Optimization instruction:

Loop unrolling: Transform loops to reduce the overhead of loop control code.

Inline expansion: Replace function calls with the function body to eliminate call overhead.

Constant folding: Compute constant expressions at compile time to reduce runtime computations.

Dead code elimination: Remove code that does not affect the program results to optimize performance.

If you find you cannot utilize the above optimization instruction, please try to optimize the program by yourself.

By fine-tuning the model with these instructions, it gains prior knowledge of optimization strategies, enabling it to generate more efficient code.

The inclusion of optimization instructions guides the LLM to focus on specific transformations that improve code performance. The fine-tuning process adjusts the model parameters θ to increase the probability of generating optimized code given an instruction. The updated probability distribution becomes:

$$P(O \mid I;\theta^*) = \prod_{t=1}^{T} P\left(o_t \mid I, o_{<t};\theta^*\right), \tag{11}$$

where θ^* represents the fine-tuned model parameters. Here, the instruction I is treated as prior knowledge and is not subjected to next-token prediction. Instead, the model focuses on predicting the next token of the output O conditioned on both the instruction I and the previous output tokens $o_{<t}$.

Training details: We trained CodeOPT with a batch size of 16 and a learning rate of 5×10^{-4} using cosine decay over 10 epochs. The maximum output length was set to 1024 tokens. For the LoRA configuration, we set the rank $r = 8$, the scaling factor $\alpha = 16$, and used a dropout rate of 0.05. Additionally, we integrated the adaptation matrices into the query and value projection layers (q_{proj} and v_{proj}) of each Transformer decoder layer.

3.4 Inference

At inference time, the fine-tuned model is employed to optimize new, unseen code snippets. Given an unoptimized code snippet $\mathbf{c}\text{in}^{\text{new}}$, the model generates the optimized code $\hat{\mathbf{c}}\text{out}^{\text{new}}$ using:

$$\hat{\mathbf{c}}\text{out}^{\text{new}} = \text{Decoder}(f\theta + \Delta\theta(\mathbf{c}_{\text{in}}^{\text{new}})), \tag{12}$$

where $f_{\theta+\Delta\theta}$ represents the transformer layers with LoRA adjustments. The decoder produces the optimized code by generating tokens sequentially, informed by the modified hidden states $\mathbf{h}'$.

The integration of LoRA brings the following advantages to the inference stage:

- **Minimal overhead:** The low-rank updates are incorporated into the model weights, adding negligible computational cost.
- **Real-time optimization**: Developers can receive immediate suggestions for code optimization as they write code.

4 Experimental setups

In this section, we present the experimental setup used to evaluate the performance of CodeOPT in comparison with state-of-the-art approaches. We begin by outlining the research questions guiding the evaluation of CodeOPT. Next, we provide details about the dataset used in our experiments and the baselines selected for comparison. We then describe the metrics employed to assess the performance of both CodeOPT and the baselines. Finally, we conclude with a description of the experimental environment.

4.1 Research questions

Our work focuses on the following two research questions (RQ):

- **RQ1 - Effectiveness** How effective is CodeOPT when compared with the baseline approaches when minimizing the code changes?
- **RQ2 - Ablation study** To what extent do our proposed training methods impact the performance of CodeOPT?

The goal of RQ1 is to evaluate the performance of CodeOPT in optimizing C/C++ programs. To assess its effectiveness, we compare it with state-of-the-art approaches, including SUPERSONIC, ChatGPT-3.5-Turbo, and GPT-4. This comparison enables us to comprehensively measure CodeOPT 's optimization capabilities. RQ2 aims to investigate whether our proposed training strategies-LoRA and instruction fine-tuning-enhance the model's ability to optimize code.

4.2 Dataset and baselines

4.2.1 Dataset

We utilize the open-source dataset released by SUPERSONIC [1], which is carefully curated from programming competition platforms such as Codeforces, AIZU, and AtCoder. These platforms were selected for their focus on optimization-centric challenges, providing a rich source of high-quality program pairs for analysis. Each sample consists of a program pair, x_n and x_{n+1}, where x_n is the base program and x_{n+1} is an optimized version authored by the same user, ensuring that the improvements are deliberate and meaningful (Table 1). The dataset guarantees

Table 1 The statistics of the dataset

Platform	Train	Evaluatoin	Test
Codeforces	276,714	877	300
AtCoder	28,665	96	–
AIZU	7497	27	259
Overall	312,876	1000	559

correctness by including only submissions accepted by the respective competition platforms. It captures key performance metrics, such as execution time, memory usage, and programming language, to track optimization progress. Furthermore, the dataset prioritizes minimal yet effective optimizations, restricting changes to no more than 20 lines and requiring a string similarity score of at least 0.8. This meticulous design enables the SUPERSONIC model to learn precise and efficient source code optimizations.

The test dataset comprises 559 unique program submissions, with 300 from Codeforces and 259 from AIZU, covering a range of problem difficulties and focusing on execution time and memory usage optimization. These problems are particularly challenging for large language models, with GPT-4 performing in the bottom 5% on Codeforces tasks. This makes the dataset well-suited for benchmarking code optimization models. To ensure unbiased evaluation, the test dataset does not overlap with the training or pre-training data.

4.2.2 Baseline selection for RQ1

We select the following three approaches as baselines to evaluate the performance of CodeOPT:

- **Supersonic** [1]: A novel code optimization model designed to learn and generate efficient source code improvements. By leveraging curated program pairs from competitive programming platforms, SUPERSONIC captures fine-grained optimizations with minimal code changes. The model aims to bridge the gap between human-level optimizations and automated systems, addressing the challenges posed by performance-centric coding tasks.
- **ChatGPT-3.5-Turbo** [15]: It is a cost-efficient large language model developed by OpenAI, designed for conversational tasks with improved speed and performance over previous models. While it offers fast responses and solid general-purpose capabilities, it may struggle with more complex reasoning and nuanced tasks compared to advanced models.
- **GPT-4** [15]: It is one of OpenAI's most powerful models, known for its improved reasoning, problem-solving, and contextual understanding. It demonstrates significantly better performance on challenging tasks, such as coding and complex language problems, but at the cost of higher latency and computational resources.

To better utilize ChatGPT-3.5-Turbo and GPT-4, we use two different prompts released by Chen et al. [1], which are tailored for source code optimization. One is a basic prompt that requires optimizing the given C/C++ programs, and another one is a Chain-of-Though [37] prompt that guides the model to generate optimized code step by step. By comparing CodeOPT against these state-of-the-art baseline approaches, we could perform an overall evaluation of the effectiveness of CodeOPT.

4.3 Evaluation metrics

The performance of the SUPERSONIC [1] is assessed using several key metrics that evaluate both the effectiveness and efficiency of the generated code optimizations. These metrics are designed to measure improvements in execution time and memory consumption, two critical aspects of program performance. Hence, we use the same metrics to evaluate the performance of all models. The first metric, **Percent Optimized** (%OPT), captures the percentage of programs in the test set that demonstrate improvements after optimization. This metric reflects the ability of the model to produce functional code modifications that positively impact performance. For this metric, only optimizations that yield at least a 20% improvement are considered valid, mitigating the effects of noise and ensuring meaningful performance gains. Another essential metric is **Performance Improvement** (PI), which quantifies the average relative improvement between the original and optimized programs. This metric is expressed as the ratio of the original resource usage (execution time or memory) to that of the optimized program. By focusing on the best optimization result among multiple predictions, the PI metric highlights the model's potential to generate highly effective code changes. These two metrics, when combined, offer a comprehensive picture of the model's ability to enhance performance while maintaining the program's correctness.

String similarity is employed to further analyze the nature of the optimizations made by all approaches. This metric, calculated using the SequenceMatcher algorithm from the `difflib` Python library, measures the degree of similarity between the original and optimized programs. The similarity score ranges from 0.0 to 1.0, with higher values indicating minimal changes. This metric helps distinguish between targeted optimizations and full program rewrites, ensuring that each approach produces concise modifications. Only program modifications with a similarity score above 0.8 are included, emphasizing targeted improvements while filtering out unnecessary rewrites.

4.4 Experimental environment

In this study, all experiments were performed on a deep learning server equipped with four NVIDIA Tesla A100 GPUs, each providing 40 GB of memory. The implementation and training of CodeOPT leveraged several Python packages, including PyTorch (version 2.3.0) [38], Transformers (version 4.45.2) [39], and Datasets (version 3.0.1) [40]. For SUPERSONIC, we utilized the model available on the Zenodo.[2] ChatGPT and GPT-4 were accessed via OpenAI's APIs. During testing, we use a beam search algorithm with size 10 to generate 10 optimized candidates for each sample in the testing set. Afterward, we manually submitted the optimized programs to relevant coding competition platforms to retrieve their runtime and memory usage, and subsequently computed various metrics locally. Besides, these

[2] https://zenodo.org/records/10889066.

Table 2 Comprehensive performance comparison across Codeforces (CF) and AIZU datasets measuring Running Time (RT) and Memory (Mem) optimization. The evaluation metrics include %OPT (Percentage of programs successfully **Opt**imized) and PI (**P**erformance **I**mprovement factor)

Model	CF (RT)		CF (Mem)		AIZU (RT)		AIZU (Mem)	
	%OPT	PI	%OPT	PI	%OPT	PI	%OPT	PI
GPT-3.5 (basic)	12.0	3.80×	3.7	11.57×	4.6	**6.87**×	1.9	3.69×
GPT-3.5 (adv)	0.7	2.86×	0	0.00×	0.8	3.11×	0.8	1.43×
GPT-4 (basic)	4.0	2.73×	1.3	1.53×	1.5	4.88×	0.8	2.45×
GPT-4 (adv)	7.7	**4.46**×	5.3	**83.07**×	3.1	3.17×	0.8	**3.79**×
Supersonic	26.0	2.65×	8.0	1.81×	3.5	2.82×	1.2	1.23×
CodeOPT	**33.7**	3.36×	**11.0**	6.23×	**6.9**	3.52×	**2.7**	2.98×

The bold values represent the highest performance improvement (PI) values and exceptional results in their respective categories

coding competition platforms could also help us check the correctness of our submitted programs, and any changes to the actions of original programs would make the optimized program failed in the testing.

5 Evaluation

5.1 Answer to RQ1: Effectiveness comparison

Table 2[3] presents a comprehensive performance analysis comparing CodeOPT against state-of-the-art baseline approaches. Our evaluation employs two key metrics: (1) %OPT, which measures the percentage of programs successfully optimized to achieve optimal performance, and (2) PI, which quantifies the factor of performance improvement achieved through optimization. The experimental results demonstrate that CodeOPT consistently outperforms all baseline models in terms of %OPT and achieves competitive performance on PI across different datasets. It is important to emphasize that the 33.7% optimization rate shown in Table 2 represents improvements achieved *beyond what compilers can automatically optimize*. Our approach targets high-level algorithmic and data structure optimizations that require semantic understanding of the code, complementing rather than replacing compiler-level optimizations. These source-level improvements provide both performance benefits and educational value for developers by demonstrating optimization patterns that can be applied in future work.

[3] Our preliminary experiments show that Code LLaMA cannot effectively perform this task with zero-shot prompting, hence we discarded results of Code LLaMA.

On the Codeforces dataset, CodeOPT achieves remarkable improvements over the strongest baseline, SUPERSONIC. For running time optimization, CodeOPT demonstrates a substantial lead of 7.7 percentage points in %OPT (33.7% vs. 26.0%), successfully optimizing an additional 23 C/C++ programs. In terms of memory optimization, CodeOPT maintains its superiority with a 3 percentage point advantage (11.0% vs. 8.0%), effectively optimizing 9 more programs than SUPERSONIC. While GPT−-3.5-Turbo (basic) and GPT-4 (advanced) achieve higher PI values in certain scenarios, it is crucial to note that CodeOPT accomplishes competitive results with at least three orders of magnitude fewer trainable parameters, demonstrating superior parameter efficiency.

The AIZU dataset results further reinforce CodeOPT 's effectiveness, where it outperforms SUPERSONIC by 3.4 percentage points in %OPT for running time optimization (6.9% vs. 3.5%). In terms of performance improvement, CodeOPT achieves a PI of 3.52× compared to SUPERSONIC's 2.82×, representing approximately a 25% higher improvement factor. For memory optimization, CodeOPT shows even more substantial gains, with 2.7% %OPT compared to SUPERSONIC's 1.2%, and a PI of 2.98× versus 1.23×, representing a 142% improvement. Notably, CodeOPT demonstrates remarkable consistency by optimizing approximately twice the number of programs compared to SUPERSONIC, highlighting its robust generalization capabilities across different program types and optimization scenarios.

5.2 Analysis of code similarity

Figure 5 illustrates the distribution of similarity scores between original and optimized programs across all approaches. As shown, both CodeOPT and SUPERSONIC maintain consistently high similarity scores (mostly above 0.8), indicating their effectiveness at producing targeted optimizations that preserve most of the original code structure. This aligns with our goal of minimizing code changes while improving performance. In contrast, GPT−-3.5 and GPT-4 exhibit a more uniform distribution across the similarity spectrum, suggesting they often generate substantially different programs rather than focused optimizations. These models frequently produce complete rewrites (similarity scores below 0.4), which may explain their occasionally higher PI values but lower consistency in delivering practical optimizations. CodeOPT demonstrates the best balance between maintaining code familiarity and achieving performance improvements, making it more suitable for real-world code optimization tasks where developers prefer recognizable, maintainable code.

5.3 Parameter-efficient fine-tuning results

Our experiments demonstrate the significant advantages of LoRA for parameter-efficient fine-tuning in code optimization tasks. As shown in Table 3, CodeOPT with LoRA achieves 30.3% optimization rate for runtime and 8.3% for memory on the Codeforces dataset, substantially outperforming the full-parameter approach (21.7% and 6.0%, respectively).

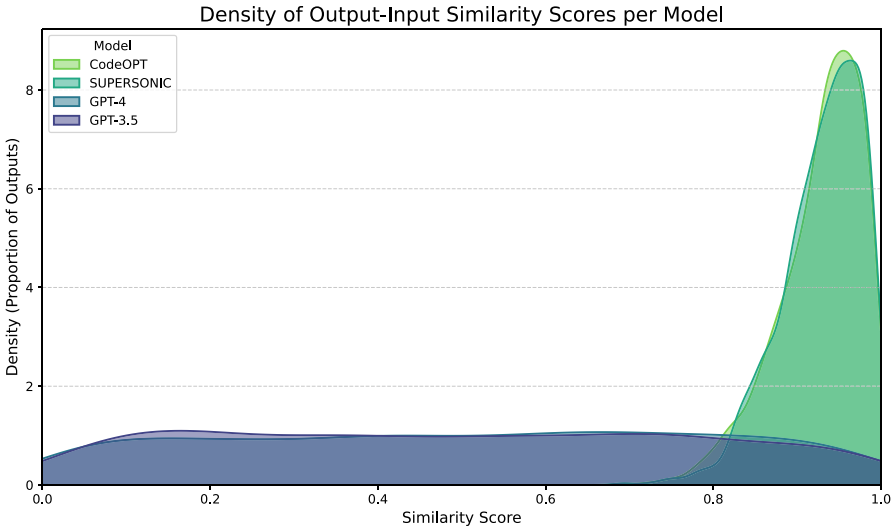

Fig. 5 Density distribution of output-input similarity scores across different models. CodeOPT and SUPERSONIC consistently produce optimized programs with high similarity to input (> 0.8), while GPT-3.5 and GPT-4 generate more varied outputs including complete rewrites

These performance gains are achieved while fine-tuning only 4–5 million parameters (less than 0.1% of LLaMA 3's seven billion parameters), resulting in three experimentally verified benefits: (1) more effective parameter updates with our 300,000 training examples, (2) reduced computational resource requirements during training, and (3) improved domain adaptation when combined with instruction-based fine-tuning. The full CodeOPT model, integrating both LoRA and instruction-based fine-tuning, further enhances performance to 33.7% for runtime and 11.0% for memory optimization.

Our results confirm that while our dataset would be insufficient for full-parameter fine-tuning of large-scale models like LLaMA 3, the parameter-efficient approach

Table 3 Ablation study results demonstrating the impact of different training strategies on model performance

Model	CF (RT)		CF (Mem)		AIZU (RT)		AIZU (Mem)	
	%OPT	PI	%OPT	PI	%OPT	PI	%OPT	PI
CodeOPT (full-param)	21.7	2.13×	6.0	1.16×	1.7	1.46×	0.3	0.99×
CodeOPT (instr)	29.0	2.87×	8.0	4.31×	3.3	2.81×	1.0	1.65×
CodeOPT (LoRA)	30.3	3.02×	8.3	4.79×	3.7	3.14×	1.3	2.07×
CodeOPT	**33.7**	**3.36×**	**11.0**	**6.23×**	**6.9**	**3.52×**	**2.7**	**2.98×**
SUPERSONIC	26.0	2.65×	8.0	1.81×	3.5	2.82×	1.2	1.23×

The bold values represent the highest performance improvement (PI) values and exceptional results in their respective categories

enables effective learning of domain-specific optimization knowledge, as evidenced by the higher Performance Improvement (PI) metrics across all test datasets.

5.4 Answer to RQ2: ablation study

To validate our design choices and quantify the contribution of each component, we conducted a comprehensive ablation study presented in Table 3. We evaluate three variant configurations of CodeOPT: (1) using only full-parameter fine-tuning, (2) incorporating instruction-based fine-tuning, and (3) employing LoRA-based parameter-efficient fine-tuning.

The results provide several key insights:

1. **Full-parameter fine-tuning limitations**: The baseline full-parameter approach performs notably worse than SUPERSONIC, confirming our hypothesis that limited training data is insufficient for effective full-parameter fine-tuning of large-scale LLMs.
2. **Instruction-based fine-tuning benefits**: The addition of instruction-based fine-tuning yields substantial improvements, enabling the model to surpass SUPERSONIC's performance. This demonstrates the value of incorporating explicit optimization knowledge into the training process.
3. **LoRA's effectiveness**: The integration of LoRA-based parameter-efficient fine-tuning provides consistent performance improvements across all metrics, validating its effectiveness for specialized code optimization tasks.
4. **Synergistic effects**: The full CodeOPT model, combining two components, achieves the best performance across all metrics, establishing new state-of-the-art benchmarks in C/C++ code optimization.

These results demonstrate that our proposed combination of parameter-efficient fine-tuning techniques and instruction-based learning creates a powerful synergy, enabling CodeOPT to achieve superior performance while maintaining computational efficiency.

6 Threats to the validity

This study faces two potential internal threats. The first is the challenge of determining the optimal configuration of LoRA parameters. To address this, we adopted the settings established in previous research [12], where these configurations demonstrated state-of-the-art performance on a specific task. By leveraging these proven settings, we mitigate concerns about internal validity. The second threat pertains to the stability of instruction fine-tuning. To address this, we closely followed the settings from prior works [4, 41] that have successfully applied instruction fine-tuning. By adopting these well-established settings, we mitigate concerns about ineffective training.

The external validity, or generalizability, of our study is a concern due to the nature of CodeOPT. Since CodeOPT is trained on data from a code competition platform, questions arise regarding its applicability in real-world scenarios. Nevertheless, companies can mitigate this limitation by training customized LLMs on their proprietary data using our approach, allowing them to effectively deploy our methodology in their specific context.

7 Conclusion

In this study, we introduce CodeOPT, a novel LLM-based approach for C/C++ source code optimization. Rather than employing full-parameter fine-tuning on an encoder-only LLM, we opt to train an optimization adapter for a decoder-only LLM. This approach allows us to train only a small subset of the original model's parameters while maintaining consistency between the pre-training and fine-tuning phases. Furthermore, to enhance CodeOPT 's ability to generate high-quality optimizations, we extend the process to instruction-based fine-tuning by incorporating standard C/C++ optimization strategies as information augmentation to the original input. This method helps guide the model in learning to apply effective optimization techniques to various programs. To evaluate the effectiveness of CodeOPT, we conducted comprehensive experiments comparing it against state-of-the-art approaches for source code optimization, including ChatGPT−-3.5-turbo and GPT-4. Our results demonstrate that CodeOPT significantly outperforms all baseline approaches, producing optimized code for a broader range of programs. In future work, we plan to explore LLM-based optimization beyond runtime and memory metrics. We will analyze the optimization patterns learned by CodeOPT to better understand its decision-making process. A key direction will be extending our approach to identify hardware-specific performance bottlenecks such as branch mispredictions, vectorization opportunities, cache misses, and prefetching issues—areas typically challenging for high-level programmers to recognize but critical for performance optimization.

Acknowledgements We are grateful to the anonymous reviewers for their valuable comments.

Author contributions Caixu Xu contributed to conceptualization, methodology, software, and writing-original draft; Hui Guo contributed to supervision and project administration, funding acquisition; Caicun Cen contributed to software, validation, and formal analysis; Minglang Chen contributed to data curation and investigation; Xiongjie Tao contributed to resources and investigation; and Jie He contributed to supervision, project administration, writing - review & editing. All authors have read and agreed to the published version of the manuscript.

Funding This work was supported by the National Natural Science Foundation of Guangxi under Grants (2025GXNSFAA069497, 2025GXNSFAA069688), in part by the Basic Ability Improvement Project for Young and Middle-aged Teachers in Guangxi, China (2024KY0694), in part by Science and Technology Development Fund, Macau SAR (0009/2024/ITP1), in part by the Key Research Project of Wuzhou University (2023B001).

Data availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Chen Z, Fang S, Monperrus M (2023) Supersonic: Learning to generate source code optimisations in c/c++. arXiv preprint arXiv:2309.14846
2. Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, Wang L, Chen W (2021) Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685
3. Lan Z, Chen M, Goodman S, Gimpel K, Sharma P, Soricut R (2020) Albert: A Lite BERT for self-supervised Learning of Language Representations
4. Zhang R, Han J, Liu C, Gao P, Zhou A, Hu X, Yan S, Lu P, Li H, Qiao Y (2023) Llama-adapter: efficient fine-tuning of language models with zero-init attention. arXiv preprint arXiv:2303.16199
5. Gu X, Zhang H, Kim S (2018) Deep code search. In: Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 933–944
6. Fang S, Tan Y-S, Zhang T, Liu Y (2021) Self-attention networks for code search. Inf Softw Technol 134:106542
7. Fang S, Zhang T, Tan Y-S, Xu Z, Yuan Z-X, Meng L-Z (2022) Prhan: Automated pull request description generation based on hybrid attention network. J Syst Softw 185:111160
8. Fang S, Zhang T, Tan Y, Jiang H, Xia X, Sun X (2023) Representthemall: A universal learning representation of bug reports. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, pp. 602–614
9. Yuan D, Fang S, Zhang T, Xu Z, Luo X (2022) Java code clone detection by exploiting semantic and syntax information from intermediate code-based graph. IEEE Trans Reliab 72(2):511–526
10. Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: Proceedings of the IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pp. 200–20010
11. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) CodeBERT: a pre-trained model for programming and natural languages
12. Silva A, Fang S, Monperrus M (2023) Repairllama: efficient representations and fine-tuned adapters for program repair. arXiv preprint arXiv:2312.15698
13. Baudry B, Etemadi K, Fang S, Gamage Y, Liu Y, Liu Y, Monperrus M, Ron J, Silva A, Tiwari D (2024) Generative AI to generate test data generators. arXiv preprint arXiv:2401.17626
14. Shypula A, Madaan A, Zeng Y, Alon U, Gardner J, Hashemi M, Neubig G, Ranganathan P, Bastani O, Yazdanbakhsh A (2023) Learning performance-improving code edits. arXiv preprint arXiv:2302.07867
15. Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, Aleman FL, Almeida D, Altenschmidt J, Altman S, Anadkat S et al. (2023) Gpt-4 technical report. arXiv preprint arXiv:2303.08774
16. Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. J Mach Learn Res 21(140):1–67
17. Fang S, Tan Y-S, Zhang T, Xu Z, Liu H (2021) Effective prediction of bug-fixing priority via weighted graph convolutional networks. IEEE Trans Reliab 70(2):563–574
18. Roziere B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu J, Remez T, Rapin J et al. (2023) Code llama: open foundation models for code. arXiv preprint arXiv:2308.12950
19. Jiang N, Lutellier T, Tan L (2021) Cure: code-aware neural machine translation for automatic program repair. In: Proceedings of the 43rd International Conference on Software Engineering, pp. 1161–1173
20. Liu Z, Xia X, Treude C, Lo D, Li S (2019) Automatic generation of pull request descriptions. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 176–188
21. Huang Y, Zhang T, Fang S, Tan Y (2022) Deep smart contract intent detection. arXiv preprint arXiv:2211.10724
22. Huang Y, Zhang T, Fang S, Tan Y (2022) Smartintentnn: towards smart contract intent detection. arXiv preprint arXiv:2211.13670

23. Chen Y, Sun Z, Gong Z, Hao D (2024) Improving smart contract security with contrastive learning-based vulnerability detection. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–11
24. Wang D, Chen B, Li S, Luo W, Peng S, Dong W, Liao X (2023) One adapter for all programming languages? Adapter tuning for code search and summarization. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, pp. 5–16
25. Garg S, Moghaddam RZ, Clement CB, Sundaresan N, Wu C (2022) Deepdev-perf: a deep learning-based approach for improving software performance. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 948–958
26. Garg S, Moghaddam RZ, Sundaresan N (2023) Rapgen: an approach for fixing code inefficiencies in zero-shot. arXiv preprint arXiv:2306.17077
27. Wang Y, Wang W, Joty S, Hoi SC (2021) Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859
28. Touvron H, Lavril T, Izacard G, Martinet X, Lachaux M-A, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F et al. (2023) Llama: open and efficient foundation language models. arXiv preprint arXiv:2302.13971
29. Bai J, Bai S, Chu Y, Cui Z, Dang K, Deng X, Fan Y, Ge W, Han Y, Huang F et al. (2023) Qwen technical report. arXiv preprint arXiv:2309.16609
30. Dubey A, Jauhri A, Pandey A, Kadian A, Al-Dahle A, Letman A, Mathur A, Schelten A, Yang A, Fan A et al. (2024) The llama 3 herd of models. arXiv preprint arXiv:2407.21783
31. Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: Proceedings of the 28th Conference on Neural Information Processing Systems (NIPS), pp. 3104–3112
32. Sennrich R, Haddow B, Birch A (2016) Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), pp. 1715–1725
33. Ke Y, Stolee KT, Le Goues C, Brun Y (2015) Repairing programs with semantic code search (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 295–306
34. Liu J, Xia CS, Wang Y, Zhang L (2024) Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems 36
35. Shen W, Li C, Chen H, Yan M, Quan X, Chen H, Zhang J, Huang F (2024) Small LLMS are weak tool learners: A multi-llm agent. arXiv preprint arXiv:2401.07324
36. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS), pp. 5998–6008
37. Wei J, Wang X, Schuurmans D, Bosma M, Xia F, Chi E, Le QV, Zhou D et al (2022) Chain-of-thought prompting elicits reasoning in large language models. Adv Neural Inf Process Syst 35:24824–24837
38. Paszke A (2019) Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703
39. Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M et al. (2020) Transformers: State-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45
40. Lhoest Q, Del Moral AV, Jernite Y, Thakur A, Von Platen P, Patil S, Chaumond J, Drame M, Plu J, Tunstall L et al. (2021) Datasets: A community library for natural language processing. arXiv preprint arXiv:2109.02846
41. Zhao H, Andriushchenko M, Croce F, Flammarion N (2024) Long is more for alignment: A simple but tough-to-beat baseline for instruction fine-tuning. arXiv preprint arXiv:2402.04833

Authors and Affiliations

Caixu Xu[1] · Hui Guo[1,2] · Caicun Cen[1,3] · Minglang Chen[1,3] · Xiongjie Tao[2] · Jie He[1]

✉ Hui Guo
3220002921@student.must.edu.mo

✉ Jie He
mvic_hj@gxuwz.edu.cn

Caixu Xu
xucaixu@gxuwz.edu.cn

Caicun Cen
3240007163@student.must.edu.mo

Minglang Chen
minglangchen@gxuwz.edu.cn

Xiongjie Tao
2240004281@student.must.edu.mo

[1] Guangxi Key Laboratory of Machine Vision and Intelligent Control, Wuzhou University, Wuzhou 543003, China

[2] Faculty of Humanities and Arts, Macau University of Science and Technology, Macau 999078, China

[3] Faculty of Innovation Engineering, Macau University of Science and Technology, Macau 999078, China