POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

An LLM-based Reinforcement Learning Framework for Code Optimization

Laurea Magistrale in Mathematical Engineering - Ingegneria Matematica

Author: Angelo Curti

Advisor: Dr. Lara Cavinato

Co-advisor: Prof. Luca Formaggia

Academic year: 2025-2026

1. Introduction

For decades, the exponential growth of computational power has largely compensated for software inefficiencies. However, as hardware scaling approaches the physical limits of Moore's Law, performance gains depend more and more on algorithmic optimization and software design. For this reason, code optimization, namely the transformation of a program into a more efficient version without altering its functional behavior, is no longer a marginal refinement but a core engineering necessity. Improving performance directly reduces computational and energy costs and constitutes a fundamental prerequisite for economically and environmentally sustainable digital systems.

Driven by these considerations, the present research was carried out in collaboration with Eni S.p.A., a global leader in the energy sector whose operations depend extensively on large-scale numerical simulation pipelines. In such an environment, optimizing computational workloads is not merely desirable but strategically critical, as improvements in code efficiency translate directly into tangible gains in performance, cost reduction, and operational scalability.

Building on these practical motivations, we investigate the automation of code optimization through Large Language Models (LLMs). Our approach relies on fine-tuning, which consists in the adaptation of a pre-trained LLM to a specific downstream task through additional training on targeted data. This strategy preserves the broad linguistic and programming knowledge acquired during pretraining while steering the model toward specific tasks.

We apply this methodology to two open-source, code-specialized model families, Qwen2.5-Coder [1] and DeepSeek Coder [2]. Both are publicly available and can be deployed and fine-tuned on a single GPU, ensuring reproducibility and computational accessibility. The ultimate objective is to obtain LLMs capable of taking an inefficient program as input and generating a functionally equivalent yet more efficient implementation.

To this end, we introduce a new two-stage fine-tuning framework that explicitly separates structural learning from execution-aware refinement. The first stage consists of Supervised Fine-Tuning (SFT) on the ACEOB dataset[4], where the models are trained on paired inefficient–efficient code examples. This phase enables them to internalize recurring optimization patterns while maintaining strict functional cor-

rectness.
The second stage employs Reinforcement Learning Fine-Tuning (RLFT) based on Group Relative Policy Optimization (GRPO). During this phase, multiple candidate solutions are generated for each input program and evaluated using execution-based signals, such as runtime and memory usage[3, 5]. By relying on relative performance comparisons within candidate groups, the framework reduces the instability caused by noisy execution measurements and provides a learning signal more directly aligned with efficiency improvement.
Finally, we evaluate the models along three complementary dimensions—structural alignment, functional correctness, and execution efficiency—to systematically assess how the proposed framework impacts their ability to perform code optimization.

2. Background

The introduction of the Transformer architecture marked the beginning of a new phase in Natural Language Processing. Transformers provided a scalable and highly parallelizable framework for sequence modeling, enabling the training of increasingly large neural networks. This architectural breakthrough laid the foundation for the development of LLMs, which soon demonstrated remarkable capabilities across a wide range of language tasks, including text generation, summarization, translation, and question answering.
Given the structural similarities between natural language and source code, it was natural to explore whether these models could be extended to programming tasks. Early empirical results confirmed that models pretrained on large corpora containing both natural language and code were able to generate syntactically valid and functionally correct programs.
Building on these encouraging results, the next step was to move from general pretraining to task specialization. This phase relied primarily on SFT. Pretrained LLMs were further trained on curated datasets composed of prompt completion pairs, such as programming problems and their corresponding solutions. This approach enabled specialization in tasks including code completion, function synthesis, bug fixing, and documentation generation. While SFT significantly improved task specific performance, it remained constrained by the quality and diversity of labeled examples. The optimization objective was still next token prediction over reference solutions, meaning that the model learned to imitate target outputs without directly optimizing for broader properties such as robustness, efficiency, or alignment with user intent.
A substantial shift occurred with the introduction of reinforcement learning based fine tuning (RLFT). By incorporating explicit reward signals derived either from human evaluators or automated evaluation pipelines, models could be optimized beyond pure imitation. Instead of merely reproducing patterns from supervised datasets, training became the maximization of externally defined signals encoding preferences, correctness checks, or performance metrics.
In program synthesis, this paradigm proved particularly effective, as compilation results, unit tests, and execution based evaluations naturally provide objective reward signals. Approaches such as PPO-Coder [5] and CodeRL [3] showed that optimizing for test pass rates leads to measurable improvements over purely supervised strategies, enabling direct optimization for functional correctness and other quantitative objectives.
These developments have progressively transformed LLM-based coding systems from experimental prototypes into tools that are now integrated into the daily workflow of software engineers. Proprietary assistants such as GitHub Copilot, built upon the Codex architecture, as well as Claude and Gemini, represent the most prominent examples of this transition. In parallel, open-source LLMs such as DeepSeek Coder [2] and Qwen2.5-Coder [1] have shown that state-of-the-art coding performance can be achieved outside proprietary ecosystems. Their competitiveness relies on concrete methodological advances, including large-scale code-dominant pretraining, refined data mixtures, extended context modeling, and optimized scaling strategies such as depth-oriented expansion and Mixture-of-Experts architectures. On the post-training side, instruction tuning and policy-optimization techniques such as Group Relative Policy Optimization (GRPO) have further strengthened alignment and task-specific specialization.

3. Objectives and Contributions

The primary objective of this work is to investigate whether LLMs can be adapted to perform code optimization. More specifically, the goal is to develop and evaluate models capable of receiving an inefficient program as input and generating a functionally equivalent implementation with improved computational efficiency.

The work is motivated by both theoretical and industrial considerations. From a research standpoint, performance-oriented optimization remains significantly less explored than other coding tasks. From an industrial perspective, the collaboration with Eni S.p.A. highlights the strategic relevance of runtime and memory efficiency in large-scale numerical simulation pipelines, where even moderate improvements translate into measurable cost and energy savings.

To address this challenge, the thesis proposes a structured two-stage fine-tuning framework characterized by several methodological innovations.

First, unlike prior works such as ACEOB[4], which rely on multi-objective or hybrid training strategies, we deliberately adopt a pure SFT setup in the initial stage. The model is trained exclusively on inefficient–efficient code pairs using a completion-based cross-entropy loss, without auxiliary objectives. This design choice creates a controlled experimental setting that isolates the structural effect of supervision, allowing us to assess how much optimization behavior can be internalized through direct transformation learning alone.

Second, we introduce an innovative RLFT stage based on Group Relative Policy Optimization (GRPO), transferring ideas from execution-driven code generation[3, 5] to the distinct setting of code optimization. The key innovation lies in the use of group-relative advantages computed across multiple sampled candidates for the same input. By centering rewards within each candidate group, GRPO mitigates the variance induced by noisy runtime and memory measurements, stabilizing policy updates without relying on an explicit critic network.

Third, the entire two-stage pipeline is explicitly designed to operate on a single GPU. In contrast to scaling-based approaches that pursue performance through increasing model size or distributed infrastructure, this work emphasizes methodological design and training strategy.

Fourth, the framework is evaluated on model families that had not previously been studied in the context of code optimization, namely Qwen2.5-Coder and DeepSeek Coder[1, 2].

4. Methodological Framework

In this section we describe the methodological framework adopted to specialize code-oriented LLMs for the task of code optimization. The approach consists of two sequential adaptation stages, SFT and RLFT, followed by a unified inference protocol and a multi-dimensional evaluation framework. Together, these elements define the full methodological perimeter of the study.

4.1. Supervised Fine-Tuning

The first stage adapts a pretrained code-oriented LLM to learn semantics-preserving inefficient–efficient program transformations through paired supervision. Let $D = \{(x_i, y_i)\}_{i=1}^{N}$ denote a dataset of inefficient programs x_i and their corresponding optimized implementations y_i. Each program is represented as a token sequence over a vocabulary V, and the LLM is modeled as an autoregressive policy π_θ that, at each generation step t, predicts the next token conditioned on the input program and on all previously generated tokens, i.e.,

$$\pi_\theta(y_t \mid y_{<t}, x).$$

SFT minimizes the completion-based cross-entropy loss

$$\mathcal{L}_{CE}(\theta) = -\sum_{i=1}^{N}\sum_{t=1}^{T_i} \log \pi_\theta\big(y_{i,t} \mid y_{i,<t}, x_i\big),$$

encouraging the model to assign high likelihood to optimized continuations conditioned on inefficient inputs. This stage enables the internalization of recurring structural optimization patterns while preserving the broad knowledge acquired during pretraining.

4.2. Reinforcement Learning Fine-Tuning

In the second stage, the model is further adapted using execution-based signals rather than explicit target completions. RLFT treats the LLM

as a stochastic policy π_θ interacting with an environment defined by the execution process. Given an input program x, the LLM generates an optimized sequence $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_T)$ autoregressively using π_θ. At each step t, the state is defined as $s_t = (x, \hat{y}_{<t})$, the action corresponds to the generation of the next token $\hat{y}_t$, and the trajectory terminates once the full program is produced. After execution, a scalar reward is assigned. The reward is set to zero if the generated program does not pass all test cases; otherwise, when correctness holds, it is defined as

$$R(\hat{y}, x) = 1 + \lambda_{\text{speed}} \frac{T_{\hat{y}}}{T_x} + \lambda_{\text{mem}} \frac{M_{\hat{y}}}{M_x}.$$

where T_c and M_c denote respectively the execution time and peak memory usage of program c.
In this setting, the objective is to maximize the expected reward under the policy π_θ, i.e.,

$$\max_\theta \ \mathbb{E}_{x \sim \mathcal{X},\ \hat{y} \sim \pi_\theta(\cdot|x)} \big[R(\hat{y}, x) \big].$$

Since execution-based signals are sparse, delayed, and affected by measurement noise, direct policy-gradient optimization can suffer from high variance and instability.
To address this issue, we adopt GRPO, which replaces absolute reward estimation with relative comparisons among multiple candidate solutions generated for the same input. For each program x, K trajectories $\{\hat{y}^{(k)}\}_{k=1}^{K}$ are sampled and evaluated. The group-relative advantage is defined as

$$\hat{A}^{(k)} = R(\hat{y}^{(k)}, x) - \frac{1}{K} \sum_{j=1}^{K} R(\hat{y}^{(j)}, x).$$

The policy is then updated by maximizing the objective

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E} \left[\frac{1}{K} \sum_{k=1}^{K} \hat{A}^{(k)} \sum_t \log \pi_\theta\big(\hat{y}_t^{(k)} \mid \hat{y}_{<t}^{(k)}, x\big) \right],$$

which increases the likelihood of trajectories whose reward exceeds the group average while reducing variance induced by noisy execution feedback.

4.3. Inference

After training, an inference phase is conducted to evaluate model performance and enable comparison with the original pretrained models. We adopt a unified framework in which the LLM is prompted to generate an optimized version of a given inefficient program using the instruction format described below:

```
Optimize the following code
for efficiency while keeping it
functionally correct.  Return
only the optimized code.
```

The generated program is then post-processed and executed in a protected environment for performance evaluation.

4.4. Evaluation Metrics

Model performance is evaluated along three complementary dimensions: functional correctness, structural alignment, and execution efficiency.
Functional correctness is measured as the proportion of generated programs that pass all available test cases:

$$\text{Correctness} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\{\hat{y}_i \text{ passes all test cases}\}.$$

This metric ensures that performance improvements are considered only when semantic equivalence is preserved.
Structural alignment is assessed using the Isomorphic Optimal Comparison CodeBLEU (IOCCB) metric, which quantifies both similarity to efficient reference implementations and structural improvement relative to the original program.
Execution efficiency is evaluated through Relative Speedup (RS) and Relative Memory Saving (RMS), defined respectively as

$$RS = \text{AVG}\left(\frac{T_{\hat{y}}}{T_x}\right), \qquad RMS = \text{AVG}\left(\frac{M_{\hat{y}}}{M_x}\right),$$

Values greater than 1 indicate improved efficiency over the original implementation.
Together, these metrics capture semantic preservation, structural transformation quality, and measurable execution-level improvements.

5. Experimental Setup

The empirical evaluation is conducted on the ACEOB dataset, derived from competitive programming submissions on Codeforces. The raw

corpus contains multiple Python solutions per problem with execution statistics and unit tests. After preprocessing, it includes 95,359 inefficient–efficient pairs across 2,444 distinct problems. For each problem, we retain the top-1 pair with the largest execution-time gap to maximize the optimization signal. Final evaluation is carried out on 100 test samples.
We evaluate two open-source code-specialized model families, Qwen2.5-Coder and DeepSeek Coder, assessing pretrained and fine-tuned variants.
During infernece, all generated programs are executed in an isolated environment under strict constraints, and correctness and efficiency are verified through controlled repeated runs.

6. Results

6.1. SFT

6.1.1 Qwen 2.5-Coder

Table 1: Qwen 2.5-Coder results before and after SFT.

Size	IOCCB	Corr.	RS	RMS
7B (base)	35.13	44%	1.053	0.888
7B (SFT)	41.14	37%	1.049	1.039
14B (base)	35.13	44%	1.024	0.857
14B (SFT)	54.75	56%	1.047	1.036
32B (base)	38.29	61%	1.069	1.003
32B (SFT)	54.26	62%	1.022	1.030

Table 1 shows that SFT consistently improves structural alignment (IOCCB) across all scales, confirming that paired inefficient–efficient supervision effectively transfers optimization patterns into the generation policy. The largest gain is observed at 14B, while the 32B variant exhibits a saturation effect, suggesting diminishing structural returns at higher capacity.
Correctness displays a scale-dependent behavior: it degrades at 7B, improves significantly at 14B, and stabilizes at 32B. This indicates that medium-scale models strike a better balance between structural transformation and semantic preservation.
RS remains close to 1 in all configurations, reflecting the limited sensitivity of runtime measurements under short execution times. RMS instead improves consistently after SFT, particularly at intermediate scales, suggesting that structural simplifications learned during supervision translate into leaner memory usage.
Overall, the 14B configuration emerges as the most balanced variant, combining strong structural gains with improved correctness and memory behavior.

6.1.2 DeepSeek Coder

Table 2: DeepSeek Coder results before and after SFT.

Size	IOCCB	Corr.	RS	RMS
1.3B (base)	30.92	40%	1.036	0.631
1.3B (SFT)	48.75	53%	1.022	0.862
6.7B (base)	37.27	55%	1.024	0.922
6.7B (SFT)	49.67	57%	1.021	0.982
33B (base)	40.27	55%	1.017	0.927
33B (SFT)	49.67	57%	1.028	0.982

Table 2 reveals a more uniform behavior. IOCCB improves systematically at all scales, with the largest gain at 1.3B, indicating that smaller models benefit strongly from explicit supervision. At higher scales, gains plateau, suggesting refinement rather than structural reshaping.
Correctness increases consistently across all variants, unlike the small Qwen model, confirming stable absorption of optimization patterns. RMS improves after SFT in all configurations, again with diminishing returns at 33B. RS remains stable and close to 1, reinforcing the hypothesis that runtime noise limits measurable speed gains.
Overall, SFT proves consistently beneficial for DeepSeek models, though improvements saturate as scale increases.

6.2. RLFT

Table 3: Test set results after RLFT.

Size	IOCCB	Corr.	RS	RMS
7B (RLFT)	22.05	72%	1.0070	0.6443
1.3B (RLFT)	32.78	35%	1.0534	0.2799

RLFT exhibits a markedly different behavior. For DeepSeek 1.3B, structural alignment and correctness degrade relative to SFT, while memory efficiency worsens. The slight RS increase

is not supported by structural or memory improvements and is likely influenced by runtime variability.
For Qwen 7B, correctness increases dramatically, reaching 72%, but IOCCB and RMS decrease substantially. This indicates that the reward signal is dominated by functional correctness: the model converges toward safer implementations that maximize test pass rate rather than toward structurally optimized solutions.
Overall, RLFT primarily amplifies the most stable component of the reward—correctness—while failing to reliably optimize execution-level metrics.

6.3. Benchmark

Table 4: One-shot benchmark results on the test set.

Model	IOCCB	Corr.	RS	RMS
GPT-5.2	17.45	26%	2.9438	1.1346
Claude 4.6	23.00	77%	1.1436	0.9962
Gemini 3	15.76	5%	4.4611	1.2812

Closed-source models evaluated in a one-shot setting exhibit heterogeneous behavior. While Claude achieves relatively high correctness, its structural alignment remains inferior to SFT-specialized open-source models. At the same time, gains in execution metrics tend to come at the expense of correctness and IOCCB, highlighting the inherent trade-offs and instability of zero-shot performance optimization.

7. Conclusions

This thesis investigated whether LLMs can be effectively specialized for automated code optimization. Unlike standard code generation, optimization introduces additional complexity: the transformation is inherently one-to-many, and execution-based signals such as runtime and memory are sparse and noisy.
To address these challenges, we proposed a two-stage fine-tuning framework that separates structural learning from execution-aware refinement. SFT enables models to internalize recurring inefficient–efficient transformation patterns, while RLFT based on GRPO incorporates execution feedback through relative reward comparisons.
The results reveal a clear outcome. Supervised specialization consistently improves structural alignment and memory efficiency while preserving or enhancing correctness, particularly at medium and larger scales. Relative speedup, however, does not show a stable positive trend, likely due to the limited runtime variability of the benchmark rather than to the absence of learned optimization behavior. In contrast, RL primarily amplifies correctness: once test cases are reliably passed, efficiency improvements remain marginal and unstable, and structural alignment as well as memory efficiency deteriorate. This indicates that, under noisy execution signals, reward-driven optimization may shift the policy toward safer but less structurally optimized solutions
Overall, this work demonstrates that structural code optimization can be effectively acquired through SFT, whereas execution-driven RLFT remains highly sensitive to reward design and signal stability. Moving toward real-world codebases and more stable, discriminative execution signals will be essential to transform LLMs from structurally competent generators into reliable, efficiency-aware systems capable of delivering measurable impact in industrial computational settings.

References

[1] Jinze Bai, Shuai Bai, Yunfei Chu, et al. Qwen technical report, 2023.

[2] DeepSeek-AI Team. Deepseek llm: Scaling open-source language models with longtermism, 2024.

[3] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022.

[4] Yue Pan, Xiuting Shao, and Chen Lyu. Measuring code efficiency optimization capabilities with aceob, 2024.

[5] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning, 2023.