POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

An LLM-based Reinforcement Learning Framework for Code Optimization

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Angelo Curti, 10615499

Abstract: Hardware advances can no longer compensate for inefficient software, placing performance at the center of rising operational costs and energy consumption. This reality is particularly clear at Eni S.p.A., a global energy company whose operations depend on large-scale numerical simulations. In this context, computational efficiency directly affects costs and energy consumption, motivating research on automated code optimization with Large Language Models (LLMs). Despite this industrial relevance, code optimization remains underexplored. Although LLMs excel at code generation and understanding, their application to optimization is limited.

This work investigates whether code-specialized LLMs can be systematically adapted to perform code optimization. The original contribution lies in the adopted two-stage framework that explicitly decouples structural learning from execution-aware refinement. Unlike prior approaches that combine heterogeneous losses, we first apply pure Supervised Fine-Tuning (SFT) on inefficient–efficient code pairs to isolate the structural impact of supervision. Only subsequently do we introduce Reinforcement Learning Fine-Tuning (RLFT) to align the model using execution feedback. RLFT is implemented using Group Relative Policy Optimization (GRPO), which stabilizes training by averaging rewards across groups of candidate solutions.

The framework is evaluated on Qwen2.5-Coder and DeepSeek-Coder models using the ACEOB benchmark. Performance is assessed along structural alignment, functional correctness, and relative execution metrics. Results show that SFT consistently improves structural alignment and memory efficiency, with correctness gains emerging at medium and larger scales. RLFT increases functional reliability by rapidly saturating the correctness component of the reward, but tends to degrade structural alignment and memory efficiency. Notably, neither approach produces consistent runtime improvements during training, as execution-based signals are inherently noisy and unstable. These findings indicate that optimization patterns can be reliably acquired through supervision alone, whereas RL fails to achieve stable efficiency alignment.

Advisor:
Dr. Lara Cavinato

Co-advisors:
Prof. Luca Formaggia

Academic year:
2024-2025

Key-words: LLM, Code Optimization, Reinforcement Learning, Fine Tuning, Group Relative Policy Optimization

1. Introduction

Over the past decades, the rapid growth of computational power, driven by advances in hardware architecture, parallelism, and clock frequency, has largely compensated for inefficient software and often masked suboptimal implementations. However, as the pace of hardware scaling [15] slows down and energy efficiency [2] becomes a central concern, performance improvements increasingly depend on algorithmic design, memory efficiency, and careful implementation choices. For these reasons, code optimization, namely the transformation of a program into a more efficient implementation without altering its functional behavior, is no longer a purely technical refinement but a key factor in reducing computational costs, lowering energy consumption, and improving scalability. This necessity becomes particularly evident in industrial contexts such as our collaboration with Eni S.p.A., a world-leading energy company whose operations rely extensively on large-scale numerical simulation pipelines. In such environments, even moderate improvements in runtime or memory usage translate into tangible energy savings, reduced infrastructure costs, and shorter development cycles, directly contributing to the economic and environmental sustainability of large-scale computational systems.

Despite its growing strategic importance, code optimization remains largely a manual and demanding activity, requiring iterative profiling, domain expertise, and careful reasoning about algorithmic complexity and runtime behavior. Even experienced programmers rely on repeated cycles of modification and measurement, with no guarantee of substantial gains, as improvements may be marginal, input-dependent, or sensitive to subtle execution conditions. Furthermore, aggressive optimizations can compromise readability or introduce unintended bugs. This combination of technical complexity, high development effort, and uncertain return naturally motivates the search for automated approaches capable of assisting or partially replacing manual optimization workflows.

Among the most promising candidates for this role are Large Language Models (LLMs), which have demonstrated remarkable capabilities in understanding and generating source code. Originally developed for natural language processing tasks, Transformer-based architectures [59] have rapidly evolved into powerful tools for programming-related applications, including code generation, translation, completion, debugging, and refactoring. Code-specialized models such as CodeT5+ [62], StarCoder [27], DeepSeek Coder [17], and Qwen2.5-Coder [21] show that large-scale pretraining followed by task-specific fine-tuning enables models to capture structural and semantic regularities of programming languages. These advances naturally raise the question of whether LLMs can move beyond code generation and contribute to automated performance optimization.

To date, research has largely concentrated on generation [23][13][58] and understanding [18][4], leaving performance-oriented code optimization comparatively underexplored. Only a limited number of recent studies have directly addressed this problem. While execution-driven reinforcement learning has been successfully applied to code generation and refinement [31][47], systematic approaches explicitly targeting efficiency improvement remain scarce. The ACEOB benchmark [36] represents one of the first structured efforts in this direction, formalizing code optimization and providing large-scale inefficient–efficient program pairs derived from competitive programming submissions to enable execution-based evaluation. Similarly, PIE [48] introduces a curated dataset of human-verified performance-enhancing edits, investigating whether language models can learn to apply targeted efficiency-oriented transformations rather than merely generate alternative implementations.

These works demonstrate that learning performance-aware transformations is feasible; however, reported efficiency gains remain limited and often unstable, particularly when relying on execution-based signals. This difficulty reflects structural characteristics of the task itself. Unlike standard code generation, optimization imposes strict semantic preservation, admits multiple valid efficient solutions for the same program, and is typically evaluated on competitive-programming benchmarks where performance differences are subtle. Moreover, execution time and memory usage are inherently noisy measurements, especially for short-running programs, making reward signals sparse, delayed, and highly variable

Building upon these considerations, the thesis explores whether medium-scale code-specialized LLMs, trained on a clean and curated dataset, can effectively learn to generate semantics-preserving code transformations that improve execution efficiency. In particular, this work investigates how different fine-tuning paradigms influence optimization behavior and whether execution-aware training can produce consistent and measurable improvements, by systematically comparing two complementary approaches: Supervised Fine-Tuning (SFT) and Reinforcement Learning Fine-Tuning (RLFT) based on execution feedback.

The framework follows a two-stage training procedure. First, the model is fine-tuned on pairs of inefficient and optimized code, so that it learns common structural transformations that preserve correctness while improving efficiency. Through this process, the model specializes its generative behavior toward efficiency-oriented transformations without discarding the broad knowledge acquired during pretraining. In the second stage, reinforcement learning (RL) is used to further adapt the model based on execution feedback. RL is a framework in which an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards. It employs a policy that defines the agent's behavior by mapping states to actions, and iteratively updates this policy with the objective of maximizing the expected cumulative reward over time. In this context,

we employ Group Relative Policy Optimization (GRPO) [46], where multiple candidate solutions are sampled for each input program and the policy is updated based on advantages computed by centering individual rewards with respect to the group mean. By relying on relative performance differences, rather than absolute reward values, the method reduces reward variance. Since this stage starts from a model already adapted through SFT, it aims to limit the instability typically observed when optimizing directly on noisy execution signals, while still promoting measurable efficiency improvements.
The empirical analysis is conducted on the ACEOB benchmark, a dataset specifically designed for code optimization in competitive programming settings. Leveraging this established benchmark allows for direct comparison with prior work while enabling a systematic evaluation of supervised and reinforcement-based adaptation strategies. The study focuses on two open-source code-specialized model families, Qwen2.5-Coder and DeepSeek Coder, enabling a comparative analysis across different architectural designs and parameter scales under consistent experimental conditions. Model performance is assessed along three complementary dimensions: structural alignment with efficient reference implementations, functional correctness, and relative improvements in execution time and memory usage.
The thesis develop as follows: Chapter 2 provides the theoretical background, reviewing the foundations of LLM and RL. Chapter 3 formally defines the code-optimization problem and establishes the theoretical framework used in this work. Chapter 4 presents the proposed methodological approach, detailing both the supervised and reinforcement-based fine-tuning strategies. Chapter 5 describes the experimental setup, including the benchmark, models, and evaluation protocol. Chapter 6 reports and analyzes the empirical results, and Chapter 7 concludes with a discussion of the main findings and possible future developments.

2. Background

This section presents the theoretical background underlying this work. We begin with an overview of Natural Language Processing, outlining the evolution from early symbolic and statistical approaches to neural architectures. We then introduce the Transformer model, whose attention-based design enabled scalable sequence modeling and laid the foundation for modern LLMs.
Building on this architectural perspective, we discuss the main paradigms of LLMs, highlighting their structural properties and training strategies. We subsequently introduce the fundamentals of RL, which provide the conceptual framework for execution-driven policy optimization.
Finally, we examine the application of LLMs to software engineering tasks, with particular emphasis on code-related modeling, thereby connecting the general theoretical foundations to the specific optimization setting addressed in this thesis.

2.1. Natural Language Processing

Natural Language Processing (NLP) [33][51] is a subfield of Artificial Intelligence that focuses on developing computational methods to enable machines to process, interpret, and generate human language. The main goal of NLP is to bridge the gap between human communication and machine understanding, facilitating interaction between the two.
It is inherently an interdisciplinary field of research that draws concepts from linguistics, cognitive science, and computer science to address a broad spectrum of challenges related to human language. The complexity always lies in the intrinsic ambiguity and variability of human language, combined with the challenge of extracting meaningful patterns from unstructured data such as text and speech.
Over the years, dedicated research efforts have progressively refined approaches to organize, analyze, and effectively utilize human language, transforming it from an unpredictable challenge into a structured and manageable resource. This progress has laid the foundation for NLP to address a wide range of practical applications, including text classification, named entity recognition, machine translation, speech recognition, question answering, text generation, and part-of-speech tagging [1].
Today, NLP has become indispensable for managing the overwhelming volume of unstructured data generated daily. Despite its vast potential, this type of data is very difficult to process and analyze. NLP techniques provide the necessary methodologies to extract meaningful information, detect underlying patterns, and derive actionable knowledge from this data. This capacity not only enhances information handling but also enables previously unachievable applications. Among the most prominent of these applications are virtual assistants [54] and conversational agents [7], which have revolutionized the way humans interact with machines. By supporting complex tasks such as information extraction [49][37], sentiment analysis [24][26], and question answering[14][43], these systems significantly boost productivity and enrich the user experience. Leveraging advanced models and algorithms, virtual assistants and conversational agents can identify trends, filter out irrelevant data, predict outcomes based on semantic content, and generate new insights, performing these tasks with minimal human

effort.
Before discussing recent advancements in NLP, such as LLM, it is essential to first understand the foundational developments that underpin these innovations.
The history of NLP dates back to 1950 when Alan Turing introduced the concept of machine understanding of language in his seminal paper "Computing Machinery and Intelligence" [57]. In this work, Turing proposed the famous Turing Test as a measure of machine intelligence, implying that a machine's ability to interpret and generate natural language would be fundamental for demonstrating human-like intelligence. The evolution of the field can be traced through four major phases: Symbolic, Statistical, Neural, and Transformer.

2.1.1 Symbolic Phase

The first phase, focused on rule-based systems in which computers attempted to emulate human language understanding by applying explicit rules to the given data. One notable result of this early approach was the Georgetown Experiment [22] in 1954, in which a system successfully translated 60 Russian sentences into English. Another early milestone was the development of ELIZA [64] in 1964, a system designed to simulate a psychotherapist using pattern matching and reflection techniques to engage in text-based dialogues with users. Despite these successes, Symbolic NLP techniques had significant limitations. They struggled with generalization, as they relied heavily on predefined rules that could not account for the full richness and flexibility of natural language. Furthermore, the computational power of the time was insufficient to handle the complex and vast structures involved in language processing. As a result, progress in the field slowed, and NLP research largely stagnated through the 1970s and early 1980s.

2.1.2 Statistical Phase

The 1980s marked a turning point in NLP with the transition from symbolic to statistical models. During those years, researchers began to explore probabilistic methods, which enabled machines to learn language patterns from data rather than relying on predefined linguistic rules.
This transition marked the beginning of the statistical phase in NLP and paved the way for the development of language models. These models are based on the premise that natural language follows probabilistic patterns, in which each word has a specific likelihood of occurring in a given sequence.
One of the earliest significant models of this phase was the Word N-gram language model [5], which sought to predict the likelihood of a word appearing in a particular sentence based on the N preceding words. This approach, while simple, proved effective in many tasks and provided a statistical basis for processing natural language. However, the N-gram model faced severe limitations arising from its reliance on word frequency, which neglects deeper linguistic structures, such as syntax and semantics. Consequently, it could not fully capture the complexity inherent in many natural language sentences and often struggled with issues such as context dependency and the handling of out-of-vocabulary terms.
A significant advancement in understanding deeper linguistic structures came with the development of the Hidden Markov Model (HMM) [38] in the 1990s. HMMs proved to be highly effective in areas like speech recognition and part-of-speech tagging. By incorporating the concept of hidden states, representing unobservable linguistic processes, HMMs enabled more sophisticated sequence modeling, allowing the past linguistic context to influence future predictions. This breakthrough was a key turning point in NLP, as it provided a framework for tackling more complex language tasks that required a deeper understanding of context.
Alongside HMMs, maximum entropy models, conditional random fields, and skip-gram language models emerged as additional tools for encoding probabilistic dependencies and solving more advanced NLP problems. These models improved task performance by modeling relationships between linguistic elements, such as between words in a sentence or phrases in a dialogue.

2.1.3 Neural Phase

In the early 2000s, NLP entered a new phase characterized by the adoption of neural network–based models. This transition was essential to address the limitations of statistical models and was driven by two main factors: the increasing availability of computational power and the exponential growth of data fueled by the rise of the World Wide Web.
During this period, more advanced models, such as Recurrent Neural Networks (RNNs) [25] and Long Short-Term Memory (LSTM) networks [20], became foundational in NLP. Unlike traditional feed-forward neural networks, these models introduced connections that formed directed cycles, enabling them to retain an internal state or memory. This capability enabled them to capture sequential patterns and contextual information more effectively. Coupled with the use of semantic networks and word embeddings, these models significantly improved

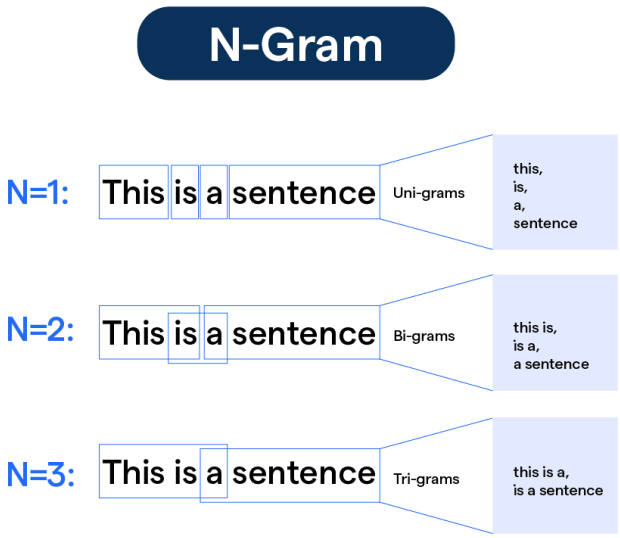


Figure 1: Illustration of an N-gram language model with a fixed context window, showing unigram, bigram, and trigram representations [61].

the ability to represent and understand deeper linguistic structures and dependencies. Another particularly notable achievement of this era was the introduction of Word2Vec [44], which revolutionized the field by encoding words as dense vector embeddings, offering a powerful way to model semantic relationships in language.
The most significant breakthrough of this era was the introduction of pre-trained language models (PLM). These models are neural networks trained on vast amounts of unlabeled language data. Those data, gathered from different sources and encompassing various text and speech types, allow the models to learn the usage of several different words and how the language is written in general, rather than being confined only to a single task. Following this initial pre-training phase, the models undergo fine-tuning, where they are specialized for specific tasks. This approach makes the models inherently general-purpose, allowing them to be adapted to new tasks with relatively little additional training on a much smaller fine-tuning dataset.
Despite the significant advancements of the Neural Network era, these models faced key limitations, such as difficulties in handling long-range dependencies due to vanishing gradient problems and the excessive time required for training caused by their inherently sequential nature, which prevented parallelization. Additionally, these models depended heavily on large amounts of labeled data, which were not always available. These challenges highlighted the need for more efficient and scalable solutions, leading to the emergence of Transformer architectures and LLMs.

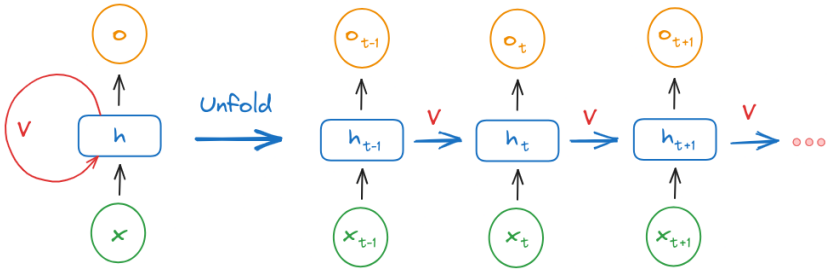


Figure 2: RNN architecture unrolled across time steps, showing input, hidden state, and output dependencies [60].

2.1.4 Transformer Phase

The proposal of the Transformer architecture marked the beginning of the current and most prolific phase of NLP. Introduced by Google in the seminal paper "Attention Is All You Need" [59], transformers revolutionized the field by overcoming recurrence in favor of the self-attention mechanism. This new approach enables the

modeling of global dependencies between inputs and outputs with unprecedented efficiency and effectiveness. To understand the potential of this invention, we will examine in detail how this architecture is structured and how each individual component functions. This will highlight the innovations and pave the way for a deeper understanding of their significance. We begin with an overview of the architecture, illustrated in Figure 3. As shown, it consists of two distinct modules: an Encoder and a Decoder.

The Encoder is formed by a stack of N_x equal layers. Each layer is composed of two sub-layers. The first is a multi-head self-attention mechanism, while the second is a simple, position-wise fully connected feed-forward network. Additionally, residual connection is applied to each sub-layer, followed by layer normalization.

Also the Decoder is composed of N_x equal layers but in addition to the two sub-layers present in each encoder layer, it inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, each sub-layer employs residual connections followed by layer normalization.

Now, to better understand how those modules work, we will trace the path of a generic input as it transforms into the model's output. We can intend our input as a text, a speech, or all of their "subsets". Without loss of generality, we will focus on a single sentence as the input. Moreover, our focus will be primarily on the attention mechanism, since it is the most significant and innovative layer of the architecture.

To start, the input is tokenized, meaning it is divided into smaller units called tokens, which can represent words or even sub-words. For simplicity, we will assume the tokens correspond to full words. The tokenized input passes through an embedding layer, which generates a high-dimensional vector representation for each token. We will refer to the dimension of those vectors as d_{model}. This process encodes unstructured data into structured, manageable entities. In the embedding space, each vector represents various characteristics of the corresponding word, and we can imagine related words to be positioned in similar directions. Here we encounter the first innovation: the mapping from tokens to the embedding space is learned from data, rather than being static, as was the case in many earlier works. This dynamic learning enables more precise representations of each token, thanks to its ability to adaptively capture context-dependent relationships and nuances in the data.

Since the model has no recurrence or convolution, we need to inject some information related to the position of each word in the sentence. To achieve this, *positional encodings* (PE) were added on top of the input embeddings at the base of both the encoder and decoder stacks. These PE layers share the same dimensionality, d_{model}, as the embeddings, allowing them to be combined through addition. There are various ways to define PE [16], including both fixed patterns and learned representations. In this work, sine and cosine functions with varying

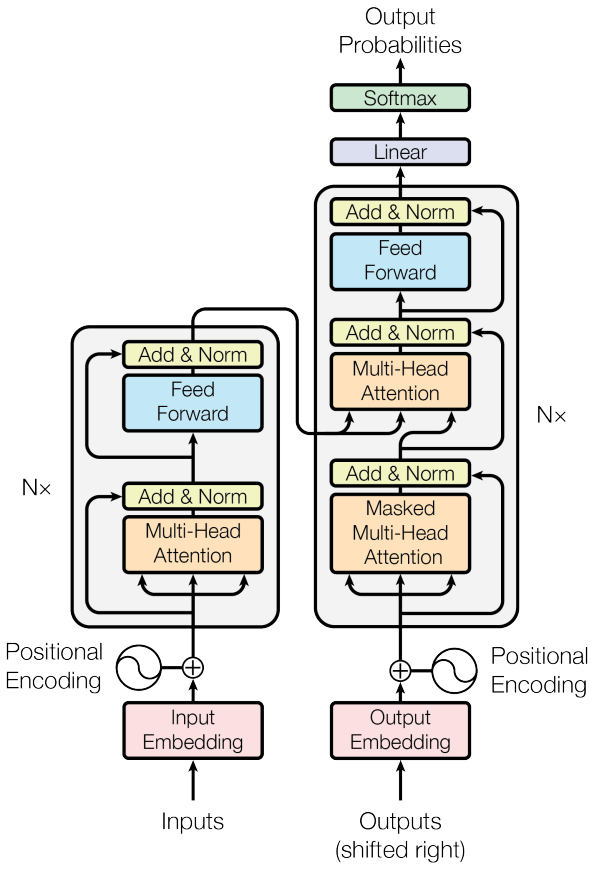


Figure 3: Transformer architecture showing the stacked encoder and decoder blocks, including multi-head attention, feed-forward layers, and positional encoding [59].

frequencies were used. Let $pos \in \{0, \ldots, L-1\}$ denote the position of a token in a sequence of length L, and let i index the embedding dimensions. We can define PE as:

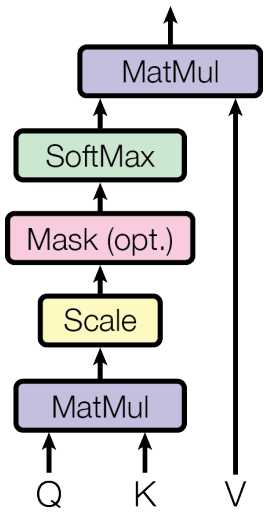
$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

$$\mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

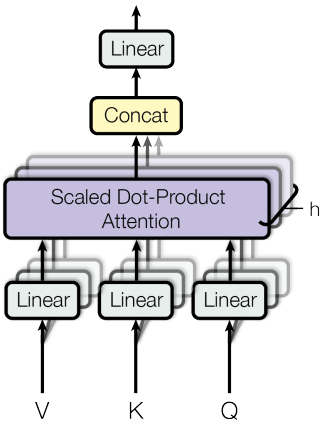
So for each embedding vector, a new vector of dimension d_{model} is generated, with a different sinusoid in each position i.
After this initial stage, the resulting vector is fed into the encoder block. Within the encoder, it passes through N_x identical modules, and for this reason we will simply explain how one of them works.
To begin, our d_{model}-dimensional vector is passed through a multi-head attention layer. As the name suggests, this layer comprises h self-attention layers. In each one, the self-attention mechanism is implemented using scaled dot-product, as shown in Figure 4a. Essentially, the attention mechanism determines the relevance of other input tokens for each token, enabling the model to dynamically weigh their contributions. In practice, this behavior is achieved by projecting the input vector into lower-dimensional spaces of dimensions d_k, d_k, d_v to obtain queries, keys, and values. Queries represent questions about the relationships between different tokens, while keys serve as the corresponding answers. Values can be seen as the information that is associated with each token and that will be transmitted and combined based on the combination of query-key pairs. The dimensionality reduction is performed using projection matrices $W_Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_K \in \mathbb{R}^{d_{model} \times d_k}$, $W_V \in \mathbb{R}^{d_{model} \times d_v}$ which, once again, are learned during the training.
Once we have queries(Q),keys(K) and values(V) the attention is computed as follows

$$\text{Attention(Q,K,V)} = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Initially, the queries are multiplied with the transposed keys to compute their dot product. The resulting values represent the relevance or compatibility of each token with the query, where high positive values indicate a strong match (positive answer), and low or negative values suggest weak or no match (negative answer). This result is then scaled to mitigate vanishing gradients, followed by the softmax function, which converts the scores into probability-like values.
Now we have the so-called attention scores, represented by a d_k-dimensional vector. The attention output is then computed by performing a dot product with the value matrix, which quantifies the actual relationship between tokens numerically.
This process is repeated for each attention head. Note that each head can be processed in parallel, resulting in substantial speedups for both inference and training. After the results from all heads are computed and gathered, they are concatenated and mapped back to the original input dimension using the matrix $W_O \in \mathbb{R}^{h \cdot d_v \times d_{model}}$.
Finally, the output of the multi-head attention layer is added to the original input vector (residual connection), and layer normalization is applied to stabilize and improve training.
Following this, the model passes through a feed-forward neural network, consisting of two linear transformations with a ReLU function between them. This sub-layer effectively combines the outputs coming from different heads and facilitates the learning of complex relationships between tokens, encouraging them to influence each other's representations.
This entire process is repeated N_x times, allowing the model to refine its understanding of complex patterns and relationships within the input. By the end of this stage, our high-dimensional vectors, enriched with detailed information about the relationships and features of each token, are ready to continue their journey into the decoder module, where they will be further processed to generate the final output.



(a) Scaled dot-product attention mechanism, where attention weights are computed as the softmax of the scaled query–key dot products.



(b) Multi-head attention mechanism, which applies multiple attention heads in parallel and concatenates their outputs.

Figure 4: Attention mechanisms in the Transformer architecture: (a) scaled dot-product attention and (b) multi-head attention [59].

As already mentioned, the decoder has a structure similar to the encoder, consisting of N_x identical layers. Each layer consists of an encoder-like structure, with a masked multi-head attention layer placed on top. The primary goal of the decoder is to generate the output, which in this context is represented by a sequence of output tokens. This is achieved by generating one token at a time, with each new token being conditioned on the previously generated tokens and the encoder's output. The masked multi-head attention layer ensures that the generation process remains autoregressive, allowing the decoder to attend only to tokens generated up to the current step. Specifically, at any step t, the decoder can access only tokens from positions 1 to $t-1$, thereby preventing future tokens from influencing the past. This constraint ensures that the decoding process respects the temporal order. We are now ready to investigate how the decoder operates at a more technical level. For each token, the output of the previous step is passed to a positional encoding layer, which acts like the one in the encoder module. After the initial pass, the generated vectors continue their journey through a masked multi-head attention layer, which operates like a standard multi-head attention layer, with one key modification in the attention mechanism. This modification prevents tokens that are currently being generated from being influenced by tokens that follow them. Specifically, it masks the lower triangular part of the attention score matrix by setting it to $-\infty$. As a result, when the softmax function is applied, these entries in the attention weights become zero. These zeros propagate through the attention block, effectively masking future tokens and ensuring that no information from future tokens can influence the generation of the current token.
The resulting vectors are fed to a multi-head attention layer in combination with the output of the encoder block. The same operations as described in the encoder section are applied here, so we won't go into further detail. However, it's worth noting the accuracy given by having two separate multi-head attention blocks. This enables us to elaborate on information in a more complex and deeper way. Moreover, this combination is not as computationally expensive as it might seem because each block can be run efficiently in parallel.
The output of the Multi-Head Attention layer is then passed through a feed-forward neural network, which operates similarly to the one in the encoder, aggregating and combining information coming from different representations, characteristics, and relationships between tokens. The process is repeated N_x times to boost performance. In the end, a linear transformation converts the d_{model}-dimensional vector into a single vector. Next, the softmax function is applied, which gives the logits for each token. This process is repeated for each token until the end of the sequence is reached. Once we have the logits, we can convert them into the desired output (text, in our case) by applying a decoding strategy, such as top_k sampling, which allows us to generate words starting from token logits.
Now that we've investigated how transformers work, we can deeply understand why this invention was so important and revolutionary. By introducing multi-head attention, transformers dramatically improve upon traditional sequential models, reducing both the training and the inference time and increasing their efficiency. The elimination of recurrent layers allows for better scalability, and the modularity of the encoder-decoder

structure enables seamless application to a wide variety of tasks, such as machine translation, text generation, and question answering.
The architecture's key innovation lies in the self-attention mechanism, which empowers the model to dynamically capture dependencies between all input tokens, regardless of their position in the sequence. This flexibility to relate tokens globally, rather than locally as done in RNNs or CNNs, brings forth more precise and adaptable language models. Furthermore, the combination of multi-head attention and PE offers an elegant solution for ensuring that context and sequence information are properly integrated throughout the architecture, while maintaining efficient computational performance.
By stacking identical layers in both the encoder and the decoder, the transformer provides a uniform yet deep structure, optimizing the model's performance through repeated transformations. The interplay between these layers, in conjunction with residual connections and layer normalization, helps stabilize training, preventing degradation over deeper architectures.
This innovative approach paves the way for even larger and more complex architectures, setting the stage for the next chapters in NLP research.

2.2. LLM

As discussed in the previous section, transformer architecture paved the way to more precise, scalable, and efficient models. This led to the rise of LLMs. LLMs mainly refer to transformer-based neural language models that contain tens to hundreds of billions of parameters, which are pre-trained on massive text data. Compared to PLMs, LLMs are not only significantly larger in model size but also demonstrate enhanced language understanding and generation capabilities. More importantly, they exhibit emergent abilities that are absent in smaller-scale language models. These include: in-context learning, where LLMs can grasp a new task from a few examples provided in the prompt at inference time; instruction following, enabling LLMs, after instruction tuning, to perform new types of tasks without requiring explicit examples; and multi-step reasoning, allowing LLMs to tackle complex problems by breaking them down into intermediate reasoning steps, as demonstrated in chain-of-thought prompting. Furthermore, LLMs can be enhanced with external knowledge, enabling them to interact more effectively with users and their environment. This opens the door to a series of different applications in which LLMs are used as generic-purpose AI agents, capable of interacting with the environment and making actions and decisions.
Now that we have explored the potential of LLMs, we can examine some of the most notable and powerful models. Grouping them into three distinct categories—encoder-decoder, encoder-only and decoder-only—helps to better understand their unique characteristics. In addition, we introduce AI agents, as they currently represent one of the most prominent and impactful application paradigms of LLMs.

2.2.1 Encoder-Decoder Architecture

Encoder-Decoder models have a complex architecture comprising both an encoder and a decoder network. The idea of using both components comes from Raffel et al.[41], which demonstrates that nearly all NLP tasks can be formulated as sequence-to-sequence generation tasks. Therefore, an encoder-decoder language model, by its very design, serves as a unified model capable of handling both natural language understanding and generation tasks. Representative encoder-decoder models include T5, PaLM [9], MASS [50], and BART [50]. T5 is one of the most prominent models in this category. It is a Text-to-Text Transfer Transformer, leveraging transfer learning to provide a unified framework in which all NLP tasks are treated as text-to-text generation challenges. Additionally, LLaMA [55], released by Meta, made a significant impact as an open-source model, often outperforming closed-source models like GPT in various tasks.

2.2.2 Encoder Architecture

The second category is represented by encoder-only models. These models were initially designed for tasks related to language comprehension, such as text classification, where the goal is to assign a class label to an input text. Notable encoder-only models include BERT and its variants, like RoBERTa [32], ALBERT [30], DeBERTa [19], XLM [29], XLNet [66], and UNILM [12]. BERT (Bidirectional Encoder Representations from Transformers) [11] is one of the most widely adopted encoder-only models. It comprises three key components: an embedding module that transforms input text into a sequence of embedding vectors, a stack of Transformer encoders that generate contextual representations from these embeddings, and a fully connected layer that maps the final representation vectors to one-hot vectors. BERT is pre-trained with two objectives: masked language modeling

(MLM) and next sentence prediction. After pre-training, the BERT model can be fine-tuned by adding a classification layer to handle various language understanding tasks, such as text classification and question answering.

2.2.3 Decoder Architecture

The third and final category comprises architectures built around a single autoregressive decoder. One of the most prominent families following this paradigm is the GPT series, developed by OpenAI. GPT-1 was the first to demonstrate that strong performance across a variety of natural language tasks could be achieved by using Generative Pre-Training (GPT) with a decoder-only Transformer model, trained on a broad and diverse corpus of unlabeled text in a self-supervised manner (i.e., predicting the next word or token). This was followed by fine-tuning on specific downstream tasks, often with significantly fewer samples. GPT-1 laid the groundwork for subsequent models, like GPT-2[39], GPT-3[6], GPT-4[35] and GPT-5[53] with each version building on the architecture and delivering enhanced performance across different language tasks.
While the GPT family represents the most influential lineage of decoder-only models, it is not the only proprietary system shaping the current landscape of LLMs. Other major commercial families include Gemini, developed by Google DeepMind, and Claude, developed by Anthropic. These models represent state-of-the-art systems built upon the Transformer decoder-only paradigm and have significantly contributed to advancing the capabilities of large-scale generative models.
The Gemini family is designed as a multimodal foundation model capable of processing and generating text, images, audio, and other modalities within a unified architecture. Built upon a Transformer-based autoregressive backbone, Gemini emphasizes long-context modeling, large-scale pretraining, and advanced reasoning capabilities, reaching the highest scores on most publicly available LLM-benchmarks. Furthermore, Gemini integrates optimized attention mechanisms and scaling strategies that enhance both inference efficiency and multimodal fusion.
Similarly, the Claude family developed by Anthropic is based on a decoder-only Transformer architecture optimized for large-scale text generation and structured reasoning. Claude models are particularly known for their strong emphasis on alignment, controllability, and safe deployment. In addition to supporting extended context windows, Claude integrates training methodologies focused on behavioral alignment, including reinforcement learning–based fine-tuning and constitutional AI principles. These design choices aim to improve reliability, robustness, and interpretability during complex interactions resulting. As a result, Claude has emerged as a leading proprietary alternative in the large language model ecosystem, competing at the highest level in terms of reasoning performance and deployment stability.
On the other side of the landscape, the rapid evolution of proprietary systems has been paralleled by the emergence of powerful open-source alternatives. Among the most prominent open-source families are the DeepSeek[52] and Qwen[3] series, which have demonstrated that competitive LLMs can be developed and deployed outside closed commercial ecosystems.
DeepSeek models have gained prominence in the open-source LLM landscape by offering competitive performance across a range of language and reasoning benchmarks while emphasizing broad accessibility and scalability. These models have been developed with the aim of providing powerful generative capabilities at a lower operational cost, making them attractive for both research and practical applications in natural language understanding and generation. Among the key innovations introduced by DeepSeek are architectural optimizations aimed at improving training and inference efficiency, refined scaling strategies that emphasize depth expansion over simple width enlargement, and enhanced attention mechanisms designed to reduce computational overhead while preserving strong performance. Through these design choices, DeepSeek has demonstrated that open-source models can achieve state-of-the-art results while maintaining cost-efficiency and reproducibility.
In parallel, the Qwen series has emerged as another major family of decoder-only models within the open-source landscape. Qwen models have rapidly established themselves by achieving strong performance across a broad spectrum of benchmarks, including language understanding, reasoning, coding, multilingual processing, and complex text generation, while similarly emphasizing scalability and deployment flexibility. These models are designed to serve as versatile foundation models capable of supporting both research-oriented experimentation and industrial-scale applications. Among the principal innovations introduced by the Qwen family are advancements in long-context modeling capabilities, refined large-scale pretraining strategies based on diverse data mixtures, and enhanced alignment techniques through instruction tuning to improve controllability and usability.

2.2.4 AI Agents

All these new and powerful models have a wide range of applications, driving increased adoption and research in the field. New models are released almost monthly, if not weekly, each bringing innovations in different areas.

At the same time, numerous practical solutions are being developed to integrate these models into industry and business applications. Beyond isolated task execution, recent developments increasingly position LLMs at the center of more sophisticated decision-making systems. In particular, they serve as the reasoning backbone of AI agents, namely, autonomous systems capable of perceiving environmental inputs, processing contextual information, and performing actions aimed at achieving specific goals. Unlike traditional single-turn generative models, AI agents are designed to operate in iterative loops, where outputs can influence subsequent decisions through interaction with external tools, memory modules, or dynamic environments. By combining language understanding, reasoning, planning, and tool use, LLM-based agents extend the capabilities of static text generation systems toward more adaptive and goal-oriented behaviors. This paradigm shift further expands the practical relevance of LLMs, positioning them as foundational components for increasingly complex intelligent systems.

Having provided a comprehensive overview of LLMs, we now turn to their applications within the specific domain addressed in this thesis, namely software engineering. Before doing so, however, it is essential to introduce Reinforcement Learning, which has played a key role in enabling the shift from natural language modeling to advanced code-related tasks.

2.3. Reinforcement Learning

Reinforcement Learning (RL) is a distinct paradigm within machine learning that focuses on training an agent to make sequential decisions through interaction with its environment. Unlike supervised learning, which relies on labeled datasets, or unsupervised learning, which seeks hidden patterns, RL allows an agent to learn directly from the consequences of its actions by receiving scalar rewards. Through this process of trial and error, the agent gradually learns a strategy—called a policy—that serves as a set of guidelines telling it which action to take in each situation, all in order to maximize its overall reward over time.

RL is fundamentally modeled as a Markov Decision Process (MDP) and is characterized by:

- a set of states S
- a set of actions A
- a reward function $R_a(s, s') = E[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$ that quantifies the benefit of an action
- a transition probability $P_a(s, s') = P(s_{t+1} = s'|s_t = s, a_t = a)$ that models how actions lead to state changes

Building on this, the goal of RL is to learn a policy $\pi(s, a) = Pr(A_t = a|S_t = s)$. Policy can be intended as complete decision-making logic that tells an agent what action to take in each state or situation. It's essentially a mapping from states to actions, like a guidebook that the agent follows to navigate its environment.

A basic RL agent interacts with the environment at each time step, it observes the current state and selects an action based on its policy π. Consequently, it transitions to a new state while receiving a reward that quantifies the immediate benefit (or penalty) of its action.

The training process in RL involves the agent learning to improve its policy over time. This is typically achieved through value-based methods, policy-based methods, or a combination of both (actor-critic methods). In value-based approaches like Q-learning, the agent learns a value function that estimates the expected return from each state-action pair, and then derives a policy by selecting actions that maximize this value. Policy-based methods, on the other hand, directly optimize the policy parameters to maximize expected returns, often using gradient ascent. During training, the agent repeatedly interacts with the environment, collecting experiences that are used to update its policy or value estimates. A key challenge in RL is balancing exploration (trying new actions to discover better strategies) with exploitation (using known good actions to maximize reward). This is commonly addressed through techniques like ϵ-greedy action selection or adding entropy to the policy. As training progresses, the agent's policy gradually converges toward an optimal strategy that maximizes long-term rewards.

The potential of RL is vast, as evidenced by its successful applications in domains ranging from robotics and autonomous driving to game playing and complex decision-making tasks.

More recently, this paradigm has been adapted to the context of LLMs, where the "environment" corresponds to the space of generated outputs and the "actions" are the model's predictions. In this setting, RL enables models to go beyond purely supervised learning objectives and refine their behavior with respect to higher-level criteria that are difficult to encapsulate in standard loss functions, such as human preferences, alignment with task goals, or qualitative performance measures. One of the most widely used RL approaches in LLM research is Reinforcement Learning from Human Feedback (RLHF)[28], in which a reward model is trained on human judgments of model outputs and then used to optimize the language model's policy so that it generates responses aligned with human preferences. RLHF typically involves SFT on annotated examples, followed by policy optimization using algorithms such as PPO, which constrains the magnitude of policy updates to maintain stability while seeking to improve reward outcomes. Beyond PPO-based RLHF, alternative optimization techniques have emerged. Direct Preference Optimization (DPO)[40] reframes the alignment problem by directly optimizing the

language model on ranked preference data without requiring an explicit reward model, simplifying the training pipeline and potentially improving efficiency . Other recent methods explore group-based (GRPO)[46]or preference-oriented variants of policy optimization to better capture complex qualitative criteria or reduce sample complexity. In our research, RL becomes particularly valuable when explicit supervision is limited or when the objective function is difficult to formalize. By integrating an RL-based framework with pretrained LLMs, we leverage the model's ability to explore large optimization landscapes and adapt dynamically to feedback signals that reflect aspects of code quality, performance, or user intent. This reward-driven, sequential learning strategy thus provides a principled way to guide the model toward behavior that is both functionally effective and aligned with application-specific performance goals.

2.4. Application of LLMs in Software Engineering

LLMs are well-suited to automated code-related tasks because they can parse structured sequences and learn statistical regularities from large corpora. Although source code is not a natural language, it is highly regular and closely linked to human intent and reasoning, which makes it a natural target for language-modeling approaches. Even if early LLMs were not explicitly designed for programming, they have been applied to software engineering since the first wave of large-scale Transformer models, obtaining strong results across many practical scenarios. This success has motivated a growing research effort toward code-specialized models and their integration into agentic systems, enabling tools such as Copilot[8], Codex, and Claude Code. Today, these assistants are increasingly embedded in the everyday workflow of software engineers, students, and other practitioners. While the architectural components of agents (e.g., memory, planning, and tool use) are crucial in practice, in the remainder of this section we focus primarily on the underlying models, as they are the most relevant building blocks for the work that follows. Special attention is given to open-source models, as they constitute the main target of the thesis.
Before diving into the most promising models, it is useful to first outline the range of tasks they are designed to address. A clear task categorization clarifies what these systems are capable of and how different applications are structured within the broader landscape of code-related uses of LLMs. It also helps us understand how a general-purpose language model can be adapted and specialized to effectively perform code-oriented tasks.

2.4.1 Taxonomy of Code-Related Tasks

LLMs are widely employed across a broad spectrum of automated code-related tasks, including code generation, summarization, translation, refinement, and completion. Beyond these core applications, their capabilities extend to complementary activities within the software development lifecycle, such as automated debugging, vulnerability detection, documentation generation, and API usage assistance. Although diverse in nature, all these tasks rely on the fundamental ability of LLMs to analyze, transform, and reason about structured code representations, highlighting their versatility in programming contexts.
From a theoretical standpoint, code-related tasks can be organized into three main families:

- Natural Language to Programming Language (NL-PL) – Tasks where the model translates human-readable instructions into executable code, such as code generation from natural language descriptions or code refinement.
- Programming Language to Programming Language (PL-PL) – Tasks involving transformations between different programming languages or code refactoring, such as translation.
- Programming Language to Natural Language (PL-NL) – Tasks that involve converting code into human-readable explanations, such as automated code summarization, documentation generation, or educational applications for understanding code behavior.

To effectively address code-related tasks, the predominant approach consists in adapting large pre-trained language models through task-specific fine-tuning. Rather than training models from scratch, this paradigm leverages the broad linguistic and structural knowledge acquired during pretraining and specializes it toward concrete programming objectives.
The most common strategy is SFT. In this setting, the model is trained on paired examples that explicitly represent the desired transformation. For NL–PL tasks such as code generation, the model is fine-tuned on natural language descriptions paired with their corresponding implementations. For PL–PL tasks like code translation or refactoring, training relies on aligned code pairs expressing the same logic in different forms. For PL–NL tasks such as code summarization, the model learns to map code snippets to concise natural language explanations. Through this supervised adaptation, a general-purpose language model progressively internalizes the structural regularities and semantic constraints of programming languages, moving from generic text modeling to specialized code-oriented behavior.

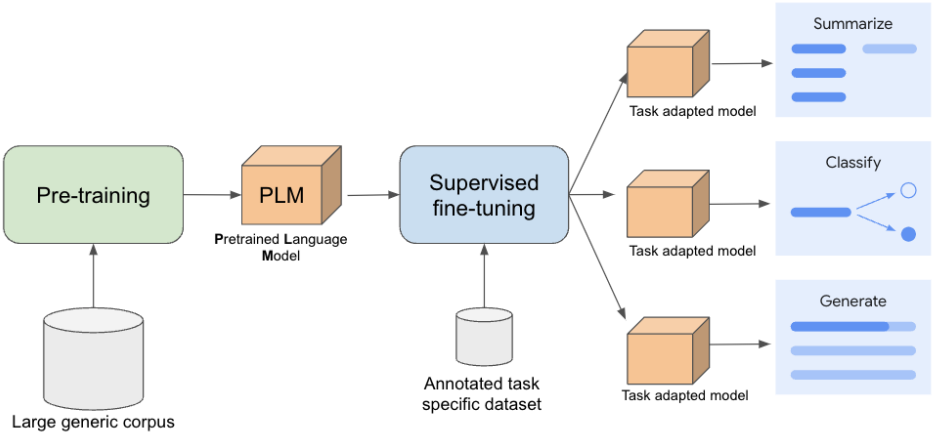


Figure 5: An overview of SFT framework[10, 65]

Beyond supervised adaptation, RL–based fine-tuning has emerged as a complementary strategy. In this setting, the model is no longer trained solely to match reference outputs, but is optimized according to a reward signal that captures higher-level quality criteria. A prominent example is Reinforcement Learning from Human Feedback (RLHF), where human preferences are used to guide the model toward more aligned and reliable outputs. In the context of code, RL can also incorporate execution-based signals, such as whether the generated program compiles, passes test cases, or satisfies performance constraints. These approaches enable the model to refine its behavior beyond imitation, optimizing for correctness, robustness, or other task-specific objectives.

2.4.2 First Generation of Code LLMs

The first groundbreaking model was GitHub Copilot, developed through a collaboration between GitHub, Microsoft, and OpenAI, and released to all developers via subscription in June 2022. Copilot serves as an AI-powered coding assistant that can be used in two primary ways: it can autocomplete code as you write, or it can generate entire code snippets based on natural language descriptions of the desired functionality. Its seamless integration into various IDEs contributed to its rapid adoption, making it a widely used tool in modern software development.

Building on this foundation, OpenAI introduced Codex, which can be considered an evolution of Copilot. Codex is a closed-source model, based on GPT-3 architecture and trained on a vast corpus of both natural language and code, achieving state-of-the-art results in code generation tasks. Notably, it demonstrated the ability to generate high-quality Python code while maintaining broad applicability across multiple programming languages and APIs, making it an incredibly versatile tool for developers.

In 2022, CodeT5 [63] marked a significant step forward in code intelligence. Built on an encoder–decoder architecture inspired by T5, it introduced code-specific pretraining objectives to better capture programming semantics. A central innovation is its identifier-aware design: source code is parsed through an AST to extract token types, enabling the model to treat identifiers (e.g., variables and function names) differently from other tokens. Pretraining includes Masked Span Prediction (MSP), Identifier Tagging (IT), and Masked Identifier Prediction (MIP), enhancing semantic understanding. Additionally, CodeT5 adopts a bimodal dual-generation objective for NL $\leftrightarrow$ PL tasks, jointly optimizing natural language and code transformations. Combined with multi-task fine-tuning, these contributions established CodeT5 as a strong benchmark for code understanding and generation.

In parallel with these task-specific innovations, classical scaling strategies continued to advance LLMs. The release of LLaMA [56] further demonstrated that compute-optimal training and carefully curated large-scale corpora could yield highly competitive and efficient architectures.

2.4.3 The Shift Toward Reinforcement Learning

Following this initial exploratory phase, the period between 2022 and 2023 marked a significant transition in the field, shifting from classical approaches toward novel methodologies designed to address two key challenges: the rigidity of learning and the lack of alignment with human behavior. The first issue stems from models traditionally following a fixed path, mapping an input to an output without adaptability or iterative refinement. The second challenge is particularly crucial in code-related tasks, where it is essential to generate outputs aligned with human expectations, avoiding errors, inefficiencies, or problematic solutions.

To address the first challenge, Salesforce introduced a novel approach to program synthesis with the CodeGen [34] models. Rather than treating it as a single-step process, these models broke down the task into smaller, more manageable sub-problems. This structured and iterative approach led to significant performance gains, allowing CodeGen to surpass existing benchmarks in the field.
For the second challenge, CodeRL [31] was the first notable model to leverage RL for better alignment. It fine-tuned language models using an actor-critic framework, where the model generated multiple candidate programs—both correct and incorrect. A separate critic component evaluated their correctness, providing feedback to refine the model's decision-making. During inference, CodeRL further improved results by analyzing failed generations, extracting error information, and using it to guide corrections, ultimately enhancing functional accuracy and reliability.
These early attempts paved the way for one of the most significant advancements in the field: the development of PPOCoder [47] by Shoajee et al. This model was built upon the CodeT5 model, fine-tuning it using the Proximal Policy Optimization [45] (PPO) algorithm. A key advantage of this RL approach was its ability to incorporate discrete feedback directly into training—something not possible in traditional frameworks due to the need for differentiable loss functions. By leveraging execution-based signals such as compilation results and runtime outputs, the model could iteratively refine its predictions based on real-world performance. To assess code quality, PPOCoder moved beyond conventional BLEU metrics, introducing a more sophisticated evaluation strategy. It employed Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs) to analyze the structural and logical correctness of generated code. Operating within this framework, PPOCoder achieved remarkable results, outperforming all competitors across a range of benchmark tests. Notably, it was designed for general-purpose code-related tasks, including code generation, translation, and summarization.

2.4.4 State-of-the-Art Code LLMs

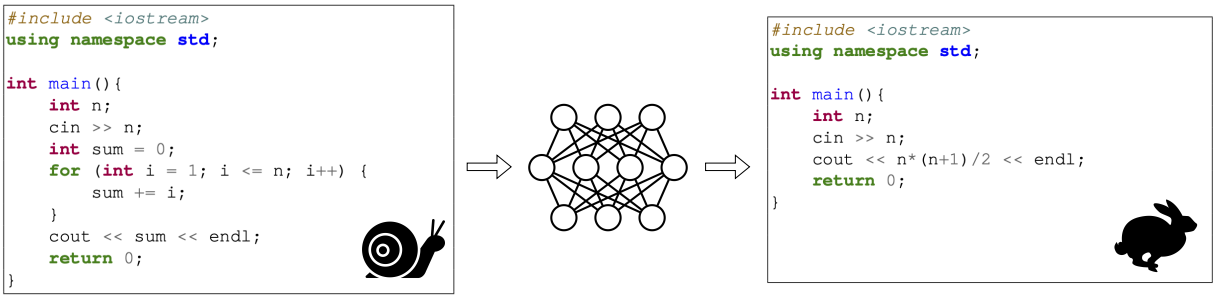
With the consolidation of large-scale pretraining, SFT, and RL-based alignment, code language models have reached a mature stage of development. The open-source ecosystem has been central to this progress, producing high-performance architectures explicitly optimized for programming intelligence, such as CodeT5+ [62] and StarCoder [27]. Among the most prominent recent examples are DeepSeek Coder and the Qwen2.5-Coder family, which combine large-scale code pretraining with strong performance across diverse programming tasks.
DeepSeek Coder[17] is a series of decoder-only Transformer models pretrained on a corpus of approximately 1.8–2 trillion tokens with a high proportion of source code, repository metadata, and related natural language contexts. Its training pipeline extends context length to up to 16 K tokens and incorporates supervised instruction finetuning, yielding both Base and Instruct variants that demonstrate strong capabilities in generation, completion, and debugging tasks across multiple programming languages and benchmarks. The focus on repository-level structure and task-aware tuning enables DeepSeek Coder to capture long-range dependencies and semantic relationships within real codebases, making it well-suited for practical software development tasks as well as research experiments in model adaptation and evaluation.
Complementing this, the Qwen2.5-Coder[21] series builds upon the broader Qwen2.5 architecture, itself trained on a massive mixed corpus, and extends it with intensive code-specific pretraining on over 5.5 trillion tokens that combine code, text-code grounding, and synthetic data. The Qwen2.5-Coder models are released in multiple parameter scales (from 0.5 B to 32 B) with instruction-tuned variants optimized for code generation, reasoning, and repair, achieving state-of-the-art open-source performance on benchmarks such as HumanEval and related tasks. The flagship 32 B model, in particular, has shown competitive accuracy and versatility, rivaling proprietary systems on several coding benchmarks and supporting execution in local environments across more than 40 programming languages.
Together, these models illustrate the current trajectory of open-source code LLM research: leveraging large and diverse datasets, extended context windows, and instruction-driven fine-tuning to narrow the gap with closed commercial offerings while providing flexible platforms for experimentation with advanced training objectives, including RL and alignment strategies.
With this comprehensive background in place, we are now well-positioned to delve into the core of this work. The following chapters will leverage these concepts to present our experimental framework, outline our methodological innovations, and report on empirical results in the optimization and adaptation of code-oriented LLMs.

3. Thesis Objectives and Contributions

In this section, we first introduce the code optimization problem at a high level, discussing its research motivations, industrial relevance, and the proposed methodological approach together with its main innovations. We then move to a formal definition of code optimization, establishing a rigorous theoretical framework that will support the developments presented in the following chapters.

3.1. Motivation and Overview

With the waning of Moore's law, optimizing program performance has become a major focus of software research. Simultaneously, LLMs have demonstrated strong capabilities at solving a wide range of programming tasks. Inspired by these advancements, this thesis aims to train an LLM to optimize code. In this context, optimization encompasses a broad range of enhancements, including reducing execution time, lowering memory usage, and improving overall resource management. The ultimate objective is to develop a model that can take a piece of code as input and output a more efficient version while preserving its original functionality.



(a) Slower Code. (b) Faster Code.

Figure 6: A simple example of code optimization [48].

This research is motivated by both theoretical insights and practical industrial needs. From a theoretical perspective, as detailed in 2.2, LLM have rapidly emerged as powerful tools in code-related tasks. Although significant advances have been made in fields like code generation and understanding, code optimization remains a relatively uncharted territory, with only a two notable studies reported in the past years [36, 48]. This gap presents an exciting opportunity to introduce fresh, innovative ideas and expand our understanding of how LLMs can be leveraged to enhance code efficiency. From an industrial perspective, the thesis was developed in collaboration with ENI S.p.A., a world-leader energy company whose operations rely heavily on computational pipelines, including large-scale numerical simulations. In such settings, even moderate improvements in runtime or memory efficiency can translate into substantial reductions in computational cost and energy consumption. Faster executions shorten processing cycles and increase resource availability, while reduced energy usage directly lowers operational expenses. Automating part of the optimization workflow also mitigates the burden on developers, who would otherwise invest significant time in manual, iterative performance tuning with uncertain outcomes.

Guided by these motivations, we leverage LLM fine-tuning to achieve our goal. Fine-tuning involves taking a pre-trained model and further training it on a specific task. This technique, which is at the core of modern LLM training paradigms, has shown remarkable results across a wide range of applications. Its key advantage is that it preserves the extensive knowledge acquired during pre-training, allowing us to specialize the model in the domain of code optimization while retaining its broad understanding of natural and programming languages. Additionally, fine-tuning is significantly less computationally expensive than training a model from scratch. This is because the pre-trained model has already learned extensive linguistic and structural knowledge, requiring only targeted adjustments rather than full-scale retraining. This makes the approach both cost-effective and environmentally sustainable.

Our work, structured as a two-stage fine-tuning framework, introduces several methodological innovations.

In the first stage, unlike the ACEOB study [36], which relies on a complex multi-objective learning strategy, we deliberately adopt a pure SFT approach. The LLM is trained on inefficient–efficient code pairs using a completion-based loss function, without incorporating auxiliary tasks or additional optimization signals. By avoiding hybrid objectives, we obtain a controlled experimental setting in which the impact of supervised specialization can be assessed without interference from heterogeneous training signals.

In the second stage, inspired by [47], which applies RL fine-tuning to code generation, we transfer this idea to the different setting of code optimization. We introduce a RL fine-tuning based on Group Relative Policy Optimization (GRPO), where the model is updated using execution-derived signals such as runtime and memory measurements. The main innovation lies in the adoption of GRPO to handle the inherent noise of execution-based rewards. By relying on the average reward over a group of candidate solutions, rather than on individual outcomes, GRPO reduces variance and stabilizes training.

An additional characteristic of the proposed framework concerns its computational footprint. The entire two-stage pipeline, including both supervised and reinforcement-based adaptation, is designed to run on a single GPU. This choice contrasts with the prevailing trend in LLM research, where performance improvements are often pursued primarily through increasing computational scale. Instead, our approach emphasizes method-

ological design and training strategy, demonstrating that meaningful specialization for code optimization can be achieved without large-scale distributed infrastructures.
The last distinguishing element of this work lies in the evaluation of the proposed framework on the Qwen2.5-Coder and DeepSeek Coder model families, which had not previously been studied in the context of code optimization.

3.2. Problem Definition

In this section, we approach the code optimization problem from a more theoretical perspective. This formalization is introduced to provide a rigorous foundation for the subsequent discussions, enabling a clear starting point for understanding the problem in a systematic way.
Let $\mathbf{C}$ denote the space of executable programs, and $\mathbf{I}$ the space of all possible inputs associated with the programs in $\mathbf{C}$. For each program $c \in \mathbf{C}$, let $\mathbf{i_c} \subseteq \mathbf{I}$ represent the set of inputs corresponding to it. We define the following cost functions:

- $S : \mathbf{C} \times \mathbf{I} \to \mathbb{R}^+$, measuring execution time (e.g., in seconds),
- $M : \mathbf{C} \times \mathbf{I} \to \mathbb{R}^+$, measuring memory consumption (e.g., in megabytes),
- $R : \mathbf{C} \times \mathbf{I} \to \mathbb{R}^+$, measuring resource usage (e.g., in kWh).

Let $G = f(S, M, R)$ represent a generic function that aggregates these cost measures into a single performance metric. This function can be interpreted as a weighted combination of execution time, memory usage, and resource consumption, or as a more sophisticated function that reflects the trade-offs among these factors in a specific optimization context. More philosophically, this function encodes the performance of a given program by considering the factors we have chosen to include, while the extension to other cases remains trivial.
The code optimization problem is then formulated as finding a transformation $T : \mathbf{C} \to \mathbf{C}$ such that, for a given program $c \in \mathbf{C}$, the transformed program $c' = T(c)$ satisfies the following properties:

- $c'(i) = c(i) \quad \forall i \in \mathbf{i_c}$, i.e., the test results remain the same as the original program,
- $G(c') - G(c) > 0 \quad \forall i \in \mathbf{i_c}$, i.e., the performance of c' is improved compared to c.

This formalization encapsulates the core objective of code optimization, ensuring that the transformed program enhances performance while maintaining the correctness of its outputs. Moreover, it paves the way for what will follow, providing a solid foundation upon which we can later delve deeper into the specifics of various methods employed.

4. Methodological Framework

This chapter presents the methodological framework adopted to address the objectives of this thesis, detailing how LLMs can be specialized for the task of code optimization through fine-tuning.
We investigate two complementary training strategies. The first is Supervised Fine-Tuning (SFT), in which the model learns to transform inefficient implementations into more efficient ones using paired training examples. The second is a RL–based approach, where the model is optimized using execution-based feedback signals that jointly account for functional correctness and performance improvements. For both paradigms, we formalize the underlying problem, describe the corresponding training procedures, and discuss the key implementation choices adopted throughout the experimental analysis. The chapter also introduces the inference protocol and the evaluation framework used to assess structural alignment, correctness, and performance gains.

4.1. Supervised Fine-Tuning

Supervised Fine-Tuning (SFT) consists in adapting a pre-trained LLM to a specific downstream task by continuing its training on a labeled dataset. Through this process, the model incorporates task-specific knowledge while preserving the broad linguistic and structural representations acquired during pre-training. In practice, SFT allows the generative behavior of the model to be specialized toward domain-dependent requirements, improving both accuracy and consistency in the target task.
This technique is widely adopted because it offers a favorable trade-off between performance and computational efficiency. Compared to training a model from scratch, SFT requires significantly less labeled data and substantially lower computational resources, since the model parameters are only adjusted rather than learned anew. At the same time, it enables the integration of domain-specific conventions and constraints, making it particularly suitable for structured tasks such as code optimization. By leveraging knowledge already embedded

in the pre-trained weights, SFT provides a principled mechanism to specialize LLMs without sacrificing their general capabilities.
However, SFT also presents well-known limitations. Excessive specialization on a narrow dataset may lead to overfitting, reducing generalization beyond the fine-tuning distribution. The adaptation process can partially erode previously acquired knowledge, a phenomenon commonly referred to as catastrophic forgetting. Moreover, SFT is inherently rigid, as it relies exclusively on explicit input–output pairs and therefore cannot naturally exploit implicit or execution-based signals that fall outside the supervised data. Finally, although less demanding than full pre-training, SFT still requires non-negligible computational resources, especially when applied to medium- and large-scale models.

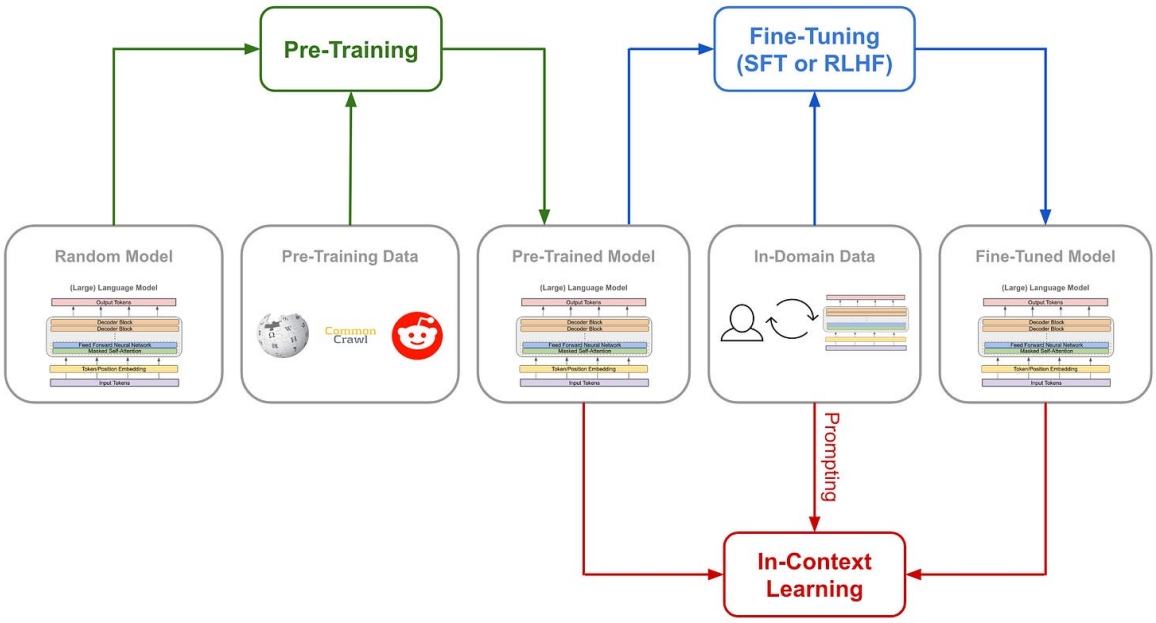


Figure 7: An overview of fine-tuning framework [65].

Now that the rationale for adopting SFT has been established, it is necessary to clarify how it is concretely formulated in our setting. The first essential element concerns the training data. Since the objective is to learn code optimization, the model must be trained on aligned pairs of inefficient and efficient implementations of the same program. Each pair represents a semantics-preserving transformation that improves performance, allowing the model to observe, in a supervised manner, how structural modifications translate into efficiency gains. The dataset construction and preprocessing pipeline are described in detail in Section 5.
The second critical component concerns the training objective, defined through the loss function. Since LLMs generate code autoregressively, the optimized program is produced one token at a time, each token conditioned on the input program and on the previously generated tokens. Consequently, we adopt a standard cross-entropy loss applied to the optimized completion, conditioned on the inefficient input. At each generation step, the model outputs a probability distribution over the vocabulary for the next token. The cross-entropy loss then compares this predicted distribution with the ground-truth target token and penalizes the model proportionally to how much probability mass it assigns away from the correct token. In this way, the model is explicitly encouraged to assign high probability to the tokens that compose the optimized implementation.
With the dataset and training objective defined, we now provide a formal description of the SFT process. Let D be a dataset containing N pairs of programs, where each pair consists of an unoptimized version and its corresponding optimized version. From now on, we will refer to the unoptimized version as x and the optimized version as y, in accordance with the classical notations commonly used in the neural network context (input x, output y). Assume that each program can be represented as a sequence of tokens selected from a vocabulary V, so a generic program C can be expressed as $C = (c_1, c_2, \ldots, c_T)$, where T denotes the number of tokens in the tokenized program. In practice, sequence lengths may vary across samples; therefore, we denote by $T^{(i)}$ the length of the i-th target sequence.
Keeping in mind the autoregressive generation process described in 2.1.4, the generative behavior of the model can be represented through a parametrized policy π_θ, which defines a probability distribution over the next token conditioned on the input program and the previously generated tokens, i.e., $\pi_\theta(y_t \mid y_{<t}, x)$. The goal of SFT is to adjust θ so that the model assigns high likelihood to the ground-truth optimized programs. Accordingly, we minimize the token-level cross-entropy loss:

$$\mathcal{L}_{\mathrm{CE}}(\theta) = -\sum_{i=1}^{N}\sum_{t=1}^{T^{(i)}} \log \pi_\theta\big(y_t^i \mid y_{<t}^i, x^i\big), \tag{1}$$

Now that we have formally introduced the SFT process in the context of code optimization, we can, finally, present the pseudocode that gives an hint on how to implement it in practice.

Algorithm 1 Supervised Fine-Tuning for Code Optimization

```
Load dataset and tokenize it
Load pre-trained model
Set-up training parameters and loss function
for i from 1 to len(dataset) do
    generate model outputs ŷ_i from input program x_i
    compute the cross-entropy loss between ŷ_i and the target y_i
    perform a backward pass to update model parameters
end for
Save weights of fine-tuned model
Perform inference to evaluate model improvement
```

4.2. Reinforcement Learning Fine Tuning

4.2.1 Motivations

Reinforcement Learning Fine-Tuning (RLFT) is a training strategy in which a pre-trained or SFT-adapted LLM is further optimized using an external reward signal instead of explicit input–output supervision. Rather than learning to imitate a reference completion, the model is trained to maximize a scalar measure of performance assigned to its generated outputs.

RLFT is particularly useful when the objective of interest cannot be naturally encoded in a supervised loss. This occurs when quality depends on external evaluation, global coherence, human preferences, or interaction with an execution environment. In such cases, imitation of a single reference solution is insufficient, as it does not guarantee improvement with respect to the true measure of performance.

The code optimization task exhibits precisely these characteristics. The transformation from inefficient to efficient code is inherently one-to-many, and the ultimate objective is not to reproduce a specific optimized implementation, but to improve execution efficiency while preserving correctness. These properties can only be assessed after full program generation and execution, and therefore lie outside the scope of a purely supervised objective.

For this reason, RLFT is introduced as a second-stage refinement after SFT. While SFT enables the LLM to internalize structural optimization patterns, it remains constrained to the distribution of reference solutions. RLFT overcomes this limitation by directly optimizing the model according to execution-based feedback, allowing fine-tuning to be guided by runtime and memory measurements under strict correctness constraints. In this way, training is aligned with the actual optimization goal rather than with imitation of a fixed target.

4.2.2 Formal Definition

Having outlined the practical motivations for adopting RLFT in the code optimization setting, we now formally describe the proposed fine-tuning methodology. By combining the RL framework introduced in 2.3 with the autoregressive formulation presented in 4.1, we characterize the LLM as an agent that, given an input program x, generates an optimized output $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_T)$ token by token over a finite horizon T. Within this setting, the elements of the RL formulation are defined as follows:

- **State:** At time step t, the environment state is defined as $s_t = (\hat{y}_{<t}, x)$, where x denotes the original source code and $\hat{y}_{<t}$ is the sequence of tokens generated so far.
- **Action:** The action at time step t corresponds to the generation of the next token, $a_t = \hat{y}_t$, selected from the model vocabulary.
- **Policy:** The stochastic policy π_θ is parameterized by the LLM and defines a conditional distribution $\pi_\theta(\hat{y}_t \mid \hat{y}_{<t}, x)$ over the next token. The policy is initialized from a pre-trained reference model ρ, such that $\pi_{\theta_0} = \rho$.

- **Trajectory:** A full trajectory $\hat{y}$ is generated autoregressively by sampling from π_θ until an end-of-sequence token is produced or a maximum length is reached.
- **Reward:** After the complete trajectory $\hat{y}$ is generated and executed, a scalar reward $R(\hat{y}, x)$ is assigned. The reward is correctness-gated: execution-based improvements are considered only if the generated code passes all functional test cases associated with x; otherwise, the reward is set to zero. Formally,

$$R(\hat{y}, x) = \begin{cases} 1 + \lambda_{\text{speed}} R_{\text{speed}}(\hat{y}) + \lambda_{\text{mem}} R_{\text{mem}}(\hat{y}), & \text{if } \hat{y} \text{ passes all test cases for } x, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

where

$$R_{\text{speed}}(\hat{y}) = \frac{T_{\text{new}}}{T_{\text{old}}}, \tag{3}$$

$$R_{\text{mem}}(\hat{y}) = \frac{M_{\text{new}}}{M_{\text{old}}}. \tag{4}$$

Here T_{old} and T_{new} denote the execution times of the original and optimized programs, while M_{old} and M_{new} denote their peak memory usage. The use of relative improvements ensures comparability across problems with different absolute runtime and memory scales, providing a scale-invariant reward signal.

Given this setting the objective of RL is to find a policy that maximizes the expected execution-based reward:

$$\max_\theta \ \mathbb{E}_{x \sim X,\, \hat{y} \sim \pi_\theta(\cdot|x)} \left[R(\hat{y}, x) \right], \tag{5}$$

where X denotes the training distribution over source programs.
To achieve this goal, we adopt a policy-gradient approach, directly optimizing the LLM policy with respect to execution-based feedback. Among policy-gradient methods, actor–critic approaches such as PPO are less reliable in our setting. These methods depend on accurate value-function estimation to compute stable advantage signals. However, in code optimization the reward is sparse, delayed until full program execution, and often noisy due to runtime variability. Moreover, multiple functionally correct solutions may exist for the same input, differing only in efficiency. For this reason, we adopt Group Relative Policy Optimization (GRPO), a policy-gradient method that avoids explicit value-function approximation. Instead of relying on a learned critic, GRPO computes advantages through relative comparisons among multiple candidate solutions generated for the same input, thereby stabilizing updates and reducing variance under noisy execution feedback.
We now formalize this optimization procedure by defining the corresponding objective, loss function, and gradient used to update the policy parameters. Given an input program x, we sample a set of K candidate trajectories $\{\hat{y}^{(1)}, \ldots, \hat{y}^{(K)}\}$ from the current policy π_θ. Each candidate is evaluated using the execution-based reward $R(\hat{y}^{(k)}, x)$. The group-relative advantage is then defined as:

$$\hat{A}^{(k)} = R(\hat{y}^{(k)}, x) - \frac{1}{K} \sum_{j=1}^{K} R(\hat{y}^{(j)}, x), \tag{6}$$

which centers rewards within the group and provides a variance-reduced learning signal.
Building on this the GRPO loss is defined as follows.

$$L_\theta^{\text{GRPO}} = \mathbb{E}_x \left[\frac{1}{K} \sum_{k=1}^{K} \hat{A}^{(k)} \sum_{t=1}^{T^{(k)}} \log \pi_\theta(\hat{y}_t^{(k)} \mid \hat{y}_{<t}^{(k)}, x) \right]. \tag{7}$$

And its gradient as:

$$\nabla_\theta L_\theta^{\text{GRPO}} = \mathbb{E}_x \left[\frac{1}{K} \sum_{k=1}^{K} \hat{A}^{(k)} \sum_{t=1}^{T^{(k)}} \nabla_\theta \log \pi_\theta(\hat{y}_t^{(k)} \mid \hat{y}_{<t}^{(k)}, x) \right]. \tag{8}$$

This loss function increases the likelihood of trajectories whose reward exceeds the group average while decreasing the probability of suboptimal ones.
Having formally defined the RLFT framework, we now present the corresponding pseudocode outlining its practical implementation.

Algorithm 2 Group Relative Policy Optimization for Code Optimization

Load and tokenize dataset X
Initialize policy π_θ from pretrained model ρ
for each training step **do**
  Sample input program $x \sim X$
  Generate K candidate solutions $\{\hat{y}^{(k)}\}_{k=1}^{K} \sim \pi_\theta(\cdot \mid x)$
  Execute each $\hat{y}^{(k)}$ and compute reward $R(\hat{y}^{(k)}, x)$
  Compute group-relative advantages $\hat{A}^{(k)}$
  Update policy parameters θ by maximizing L_θ^{GRPO}
end for
Return optimized policy π_θ

4.3. Inference

At test time, both the base and fine-tuned models are evaluated in a purely autoregressive generation setting. Given an unoptimized source program x, the model generates an optimized candidate $\hat{y}$ token by token, according to the learned conditional distribution $\pi_\theta(\hat{y}_t \mid \hat{y}_{<t}, x)$.
All models are prompted using the same instruction-based chat format, reported in Appendix B, and adopted during SFT, ensuring consistency between training and evaluation. The input consists exclusively of the unoptimized code, formatted within the instruction template, while the model is required to produce only the optimized implementation without additional explanations or natural language commentary.
For each test sample, a single candidate solution is generated, in line with the objective of developing a tool capable of performing one-shot code optimization.
After generation, the produced code undergoes a post-processing stage to ensure executability. Any extraneous formatting artifacts (e.g., Markdown code fences or unintended textual prefixes) are removed, and the resulting Python code is extracted and written into a temporary execution file.
This inference protocol is applied uniformly across base, SFT, and RLFT models, ensuring fair and consistent comparison across training paradigms.

4.4. Metrics

Evaluating code optimization models is inherently more complex than evaluating classical code generation systems. In the optimization setting, three aspects must be assessed jointly: preservation of functional correctness, structural proximity to efficient implementations, and measurable improvement in computational efficiency. These objectives capture complementary properties of the generated code and cannot be adequately represented by a single metric.
Execution-based indicators provide direct evidence of runtime and memory improvements, but they are often affected by measurement noise, particularly in competitive-programming benchmarks where execution times are extremely short. Conversely, lexical similarity metrics are poorly suited for optimization tasks, as multiple structurally distinct implementations may achieve equivalent efficiency and correctness.
For these reasons, we adopt a multi-dimensional evaluation protocol. Correctness is assessed through functional testing, efficiency is measured using relative execution-time and memory metrics, and structural alignment with optimized implementations is quantified through the Isomorphic Optimal Comparison CodeBLEU (IOCCB) metric [36]. For completeness, we introduce the execution quantities used to define the performance metrics:

- $\mathbf{T}_{\text{old}}$: execution time of the original code.
- $\mathbf{T}_{\text{opt}}$: execution time of the optimized code generated by the model.
- $\mathbf{M}_{\text{old}}$: peak memory usage of the original code.
- $\mathbf{M}_{\text{opt}}$: peak memory usage of the optimized code generated by the model.

The following subsections formally define these evaluation measures.

4.4.1 Isomorphic Optimal Comparison CodeBLEU (IOCCB)

To evaluate structural alignment in the code optimization setting, we adopt the Isomorphic Optimal Comparison CodeBLEU (IOCCB) metric [36]. To the best of our knowledge, IOCCB is currently the only structural similarity metric in the literature specifically designed for code efficiency optimization tasks.
Let x denote the original inefficient code, $\hat{y}$ the optimized code generated by the model, and $\mathcal{E} = \{e_1, \ldots, e_K\}$ the set of alternative efficient implementations provided in the benchmark.

To reduce sensitivity to superficial syntactic variations, all programs are standardized via uniform identifier renaming, producing $\tilde{y}$ and $\tilde{e}_k$. Similarity is then computed using CodeBLEU[42].
We first measure the maximum similarity between the generated solution and the efficient reference set:

$$B_{\max} = \max_{e_k \in \mathcal{E}} \text{CodeBLEU}(\tilde{y}, \tilde{e}_k).$$

Next, we compute the average similarity of both the original and generated programs with respect to the efficient reference set:

$$O_{\text{avg}} = \frac{1}{K} \sum_{k=1}^{K} \text{CodeBLEU}(x, e_k), \qquad S_{\text{avg}} = \frac{1}{K} \sum_{k=1}^{K} \text{CodeBLEU}(\tilde{y}, \tilde{e}_k).$$

The IOCCB score is defined as

$$\text{IOCCB} = B_{\max} + \sqrt{S_{\text{avg}} - O_{\text{avg}}}.$$

This formulation captures both proximity to the closest efficient implementation and structural improvement relative to the original program.

4.4.2 Correctness

The correctness metric evaluates whether the optimized code preserves the functional behavior of the original implementation. A sample is considered correct if the optimized code executes successfully without runtime errors and produces valid outputs according to the problem specification and available test cases. Correctness is reported as an average binary score over the test set of length N:

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

4.4.3 Relative Speedup (RS)

Relative speedup measures the execution-time improvement achieved by the optimized code relative to the original implementation. It is defined as the average ratio between the execution time of the original code and that of the optimized code:

$$\text{RS} = \text{AVG}\left(\frac{\mathbf{T}_{\text{new}}}{\mathbf{T}_{\text{old}}}\right).$$

Values greater than 1 indicate a speed improvement, while values below 1 correspond to slower optimized code.

4.4.4 Relative Memory Saving (RMS)

Relative memory saving evaluates the reduction in peak memory usage achieved by the optimized code. It is defined as the average relative decrease in memory usage with respect to the optimized code:

$$\text{RMS} = \text{AVG}\left(\frac{\mathbf{M}_{\text{new}}}{\mathbf{M}_{\text{old}}}\right).$$

Values greater than 1 indicate improved memory efficiency, while values below 1 correspond to increased memory usage.

5. Experiment Details

This section outlines the experimental setup adopted throughout the thesis and motivates the main design choices behind it. We begin by presenting the dataset used for training and evaluation, with particular attention to how it captures the competitive-programming setting and which limitations it introduces. Next, we describe the families of LLMs considered in our study. We then detail the evaluation protocol for code optimization, which must jointly account for functional correctness and efficiency improvements.

5.1. Dataset

The experimental evaluation conducted in this thesis is based on the ACEOB dataset[36], a benchmark specifically designed for the code optimization task. The dataset is constructed from Python solutions submitted to the Codeforces competitive programming platform and provides paired implementations of inefficient and efficient programs solving the same problem.
ACEOB originates from the ACEOB-Ori corpus, which contains raw competitive programming submissions together with extensive auxiliary information. For each problem, the dataset includes the natural language problem statement, input and output specifications, public and hidden I/O unit tests, execution time and memory constraints, as well as metadata such as difficulty levels and algorithmic tags. This information enables automatic verification of functional correctness and supports a precise assessment of performance improvements, which is central to the optimization task.

5.1.1 Preprocessing Pipeline

To ensure high data quality and suitability for LLM training, the original corpus undergoes a structured preprocessing pipeline.
The first step consists of code normalization. Raw source files are cleaned by removing unnecessary comments and special operations that are not relevant for execution. This reduces noise while preserving the functional behavior of the programs.
In the second step, an Abstract Syntax Tree (AST) analysis is applied to eliminate formatting inconsistencies and resolve potential syntactic issues that could prevent compilation or execution. This guarantees that each code sample is structurally valid Python code.
Subsequently, near-duplicate programs are removed to avoid redundancy and training bias. Similarity thresholds are computed using longest common substring techniques, and only sufficiently distinct implementations are retained.
Finally, overly long samples are filtered out by applying both relative and absolute length constraints. This ensures compatibility with the maximum input size supported by the models used during training and evaluation.
After preprocessing, the dataset consists of clean and standardized efficient–inefficient code pairs suitable for controlled experimentation.

5.1.2 Dataset Statistics

After the preprocessing stage, the ACEOB dataset contains 95,359 efficient–inefficient code pairs, of which 9,305 belong to the official test split.
Each data item consists of two functionally equivalent implementations of the same problem, one inefficient and one efficient. On average, each problem includes 1.88 public unit tests and 24.19 hidden unit tests. Additionally, each problem is associated with an average of 40.57 alternative efficient reference implementations.
The dataset covers 5,244 programming competition problems grouped into 2,444 distinct problem statements, with 289 problems included in the test split. The average length of the natural language descriptions is approximately 345 words, indicating a non-trivial level of problem complexity. This setting closely reflects realistic competitive programming scenarios and poses substantial challenges for code generation and optimization models.

5.1.3 Experimental Subset Construction

For the experimental evaluation presented in this thesis, a dedicated subset of ACEOB is constructed.
For each problem, the top-1 efficient–inefficient pair exhibiting the largest execution time gap is selected. This choice emphasizes the optimization signal and allows for a clearer assessment of the model's ability to produce meaningful performance improvements.
From the official ACEOB test split, 100 examples are further selected to conduct the final evaluation experiments. This controlled subset enables detailed analysis while maintaining computational feasibility.

5.2. Models

For what concerns LLMs, we select two open-source code language model families that can be deployed and fine-tuned on a single GPU: Qwen Coder 2.5 [21] and DeepSeek Coder [17]. The choice of open-source models that are feasible to run on commodity hardware reflects both practical constraints and the desire to work with models whose complete weights and training procedures are publicly accessible. Open-source availability facilitates reproducibility, modification of training pipelines, and direct deployment without reliance on remote

APIs or proprietary services. All training and inference hyperparameters, including optimization settings, sequence length configuration, and decoding parameters, are reported in detail in Appendix B.

5.3. Experimental Setup

The execution of model-generated code is handled in a fully isolated and controlled environment in order to ensure reproducibility, safety, and reliable performance measurements. All Python code is executed in a separate operating system process, launched via the system interpreter using a subprocess-based execution pipeline. The evaluation script dynamically constructs an execution harness that embeds the generated code, writes it to a temporary file, and invokes the corresponding process with strict execution constraints.

Input and output are handled following a competitive programming paradigm. For each test case, the input is injected through standard input (`stdin`), while the produced output is read from standard output (`stdout`) and compared against the expected result. This design closely mirrors the execution model adopted by competitive programming platforms and ensures consistency with the structure of the ACEOB dataset.

To prevent failures due to infinite loops, deadlocks, or blocking operations, multiple layers of execution control are enforced. At the process level, a global timeout is applied to the entire execution. Additionally, within each subprocess, a watchdog thread is spawned to monitor individual executions and terminate the process if a per-run timeout is exceeded. This dual-level timeout mechanism provides robust protection against non-terminating or malicious code while preserving the ability to measure valid executions accurately.

Whenever possible, user-generated code is encapsulated within a callable function and invoked directly to minimize execution overhead and improve timing stability. In cases where this is not feasible, such as when the code relies on global statements or side effects, the execution is performed using `exec` within a dedicated and isolated namespace. This approach balances execution efficiency with the flexibility required to support a wide range of code structures produced by LLMs.

To obtain reliable performance measurements, each execution is repeated multiple times. During the actual measurement phase, Python's garbage collector is explicitly disabled to minimize non-deterministic interruptions. Both wall-clock time and CPU time are recorded, and peak memory usage is tracked using `tracemalloc`, providing a comprehensive view of the runtime behavior of each solution.

The validation of functional correctness is performed through a hierarchical comparison strategy designed to handle the variability of model-generated outputs. The comparison pipeline starts with exact output matching after whitespace normalization, followed by line-by-line comparison. If necessary, numerical outputs are compared using tolerance thresholds to account for floating-point imprecision. Finally, more permissive matching strategies are applied, including unordered comparisons of lines or rounded numeric values, in order to accommodate differences in output ordering or minor numerical deviations without penalizing solutions that are functionally equivalent.

Overall, this execution and evaluation framework is designed to strike a balance between strict correctness enforcement and practical robustness. It enables accurate performance measurement while accommodating the inherent variability of automatically generated code, making it well suited for large-scale evaluation of code optimization models in competitive programming scenarios.

6. Results

In this section, we present and analyze the experimental results obtained from the two proposed fine-tuning frameworks. The objective of this evaluation is to assess the effectiveness of both supervised and reinforcement-based adaptation strategies in improving code optimization performance under the execution-aware setting described in the previous chapters.

For completeness and reproducibility, detailed training curves, including loss, accuracy, reward dynamics, and optimization diagnostics, are reported in Appendix A. Furthermore, the full set of experimental configurations and hyperparameter specifications for all runs are provided in Appendix B.

The section is organized into three main subsections. First, we report the results achieved through SFT, examining the impact of paired inefficient–efficient supervision across different model families and scales. Subsequently, we analyze the outcomes of the RL–based Fine-Tuning stage, discussing the behavior of execution-driven policy optimization and its empirical implications. Finally, we evaluate leading closed-source LLMs in a one-shot setting to establish an external benchmark and contextualize the performance of our specialized open-source models.

6.1. SFT

6.1.1 Qwen Coder 2.5

We begin our results section with the Qwen Coder 2.5 family, fine-tuning the 7B, 14B, and 32B Instruct variants. The objective of this analysis is to assess whether SFT effectively specializes the models toward code optimization, and to understand how model scale influences the balance between alignment with efficient implementations, functional correctness, and execution-level performance.

Table 1: Test set results for Qwen 2.5 Coder models before and after SFT. IOCCB (Isomorphic Optimal Comparison CodeBLEU), RS (Relative Speedup), RMS (Relative Memory Saving).

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

Table 1 highlights how SFT reshapes the behavior of the Qwen 2.5 Coder models across different scales.
The most consistent effect is observed on IOCCB. Fine-tuning leads to a clear improvement across all model sizes, confirming that the models learn to generate implementations that are structurally closer to optimized references. The gain is particularly pronounced for the 14B variant, while the 32B model reaches a comparable IOCCB level, suggesting a saturation effect at larger scales. This indicates that SFT effectively transfers recurring optimization patterns from the dataset into the generation policy, and that beyond a certain capacity the structural alignment no longer increases proportionally with parameter count.
Functional correctness, however, exhibits a scale-dependent trend. The 7B model experiences a degradation after fine-tuning, implying that limited-capacity models may apply structural transformations without reliably preserving semantic equivalence. In contrast, the 14B model shows a substantial improvement, achieving the largest correctness gain among the SFT variants. The 32B model improves only marginally with respect to its base version, suggesting diminishing returns at higher scales. Overall, this pattern points to a non-monotonic relationship between scale and reliability: medium-sized models appear better positioned to internalize optimization strategies while maintaining functional consistency.
RS remains stable across all configurations. The values are very close to 1 and no consistent trend emerges either across scales or between base and fine-tuned versions. The limited variation suggests that runtime measurements are likely influenced by the characteristics of the benchmark, where short execution times amplify measurement noise. In such a setting, structural improvements do not necessarily translate into measurable timing differences.
RMS shows a clearer evolution with scale. Larger models generally produce implementations with better memory behavior, but the improvement plateaus at higher parameter counts. While the 7B and 14B models benefit visibly from fine-tuning, the 32B variant does not extend this trend further. This suggests that the memory-related optimization patterns learned during SFT are effectively captured at intermediate scales, after which additional capacity does not yield proportional gains.
In summary, SFT consistently enhances structural alignment across the Qwen family, while its impact on correctness and efficiency depends strongly on model scale. The 14B configuration emerges as the most balanced variant, combining improved structural similarity with higher functional reliability, whereas gains at 32B indicate saturation effects rather than continued scaling benefits.

6.1.2 Deepseek Coder

We now turn to the DeepSeek Coder family, fine-tuning the 1.3B, 6.7B, and 33B variants.

Table 2: Test set results for DeepSeek Coder models before and after SFT. IOCCB (Isomorphic Optimal Comparison CodeBLEU), RS (Relative Speedup), RMS (Relative Memory Saving).

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

Looking at Table 2, consistent improvements emerge after SFT, although the gains tend to plateau as model scale increases.
Starting from IOCCB, fine-tuning produces systematic improvements across all three model sizes. The increase is particularly pronounced for the 1.3B variant, which benefits substantially from supervision, indicating that smaller models strongly leverage explicit optimization signals. The 6.7B and 33B models also improve, but the magnitude of the gain becomes progressively smaller. This suggests a scale-related plateau effect: as model capacity grows, structural alignment is already relatively strong in the base model, and SFT refines rather than fundamentally reshapes generation behavior.
Functional correctness follows a similarly coherent trend. Unlike what was observed in the smaller Qwen variant, DeepSeek does not suffer degradation at low capacity. Correctness improves consistently after fine-tuning for all scales, indicating that supervised optimization patterns are absorbed without compromising semantic validity. However, here too, a plateau emerges: the largest model shows only marginal gains relative to its already strong base performance. This suggests that beyond a certain capacity, additional supervision mainly consolidates existing representations rather than unlocking new reliability improvements.
RS exhibits the same behavior discussed previously. Values remain close to 1 across all configurations, and no consistent upward trend is observed after fine-tuning. Variations are modest and do not indicate systematic runtime acceleration. As before, this is likely attributable to benchmark characteristics, where short execution times and system-level noise obscure small but potentially meaningful structural optimizations.
RMS instead shows clearer and more stable improvements. After fine-tuning, all variants generate leaner implementations with higher relative memory savings. The improvement is more visible for smaller and mid-sized models, while the 33B variant displays a limited additional gain over its base version, again reflecting a saturation effect at higher scales. This indicates that SFT effectively encourages transformations that reduce intermediate allocations or simplify data structures, but the benefit does not grow indefinitely with parameter count.
Overall, SFT proves consistently beneficial for DeepSeek Coder across all scales. Structural alignment, functional correctness, and memory efficiency improve systematically, yet gains diminish as model size increases, revealing a clear plateau phenomenon. The limited impact on measured speedup appears once more to stem from benchmark constraints rather than from an absence of learned optimization behavior.

6.2. RLFT

We continue our analysis by examining the results of RLFT on DeepSeek 1.3B and Qwen 7B. The objective of this section is to assess whether reward-driven fine-tuning further specializes the models toward code optimization, and to evaluate how this additional optimization step affects structural alignment, functional correctness, and execution-level performance compared to the supervised baseline.

Table 3: Test set results for Qwen-7B and DeepSeek-1.3B models after RLFT. IOCCB (Isomorphic Optimal Comparison CodeBLEU), RS (Relative Speedup), RMS (Relative Memory Saving).

Model	IOCCB	Correctness	RS	RMS
Qwen-7B (RLFT)	22.05	72%	1.0070	0.6443
DeepSeek-1.3B (RLFT)	32.78	35%	1.0534	0.2799

Looking at Table 3, the test set results reveal a markedly different behavior compared to SFT, highlighting how reward-driven fine-tuning reshapes model priorities.
For DeepSeek 1.3B, RLFT leads to a clear deterioration in structural alignment and functional correctness relative to the SFT baseline. IOCCB decreases substantially, and correctness drops as well, indicating that the reward optimization process destabilizes previously learned structural optimization patterns. Memory efficiency worsens significantly, while RS shows a slight increase. However, this timing improvement is not accompanied

by structural or memory gains and is therefore unlikely to reflect genuine optimization behavior. Instead, it is plausibly influenced by runtime variability. Overall, for DeepSeek 1.3B, RLFT appears to weaken structural specialization without producing consistent efficiency benefits.
For Qwen 7B, the pattern is different but equally informative. Structural alignment decreases noticeably and memory efficiency degrades. In contrast, correctness increases dramatically, reaching the highest value observed across all configurations. This suggests that the reward signal is strongly dominated by functional correctness. Since passing test cases provides a clear and stable feedback signal, the model shifts its generation policy toward safer and more conservative implementations that maximize test success, even at the expense of structural similarity to optimized references or memory efficiency.
Overall, RLFT does not lead to consistent structural or efficiency improvements in either model. Instead, optimization converges toward the most stable and least noisy component of the reward, namely correctness. In the absence of a strong and well-shaped execution-level signal, reinforcement learning amplifies the reliability objective while neglecting structural and resource-oriented optimization. This indicates that, in the current setup, RL primarily reinforces functional validity rather than true code optimization.

6.3. Benchmark

As a final step of the experimental study, we evaluate state-of-the-art closed-source models, representing some of the strongest proprietary systems currently available on the market. Specifically, we assess their performance in a one-shot setting, meaning that each model is prompted once without any task-specific fine-tuning or adaptation. This evaluation provides an external reference point to contextualize the results obtained with our supervised and RL–based fine-tuned open-source models. The comparison is conducted using the same benchmark, execution protocol, and evaluation metrics ensuring fairness and consistency across all models.
The results highlight a heterogeneous behavior across proprietary systems. While some models achieve high functional correctness, structural alignment with efficient references (as measured by IOCCB) remains limited compared to fine-tuned open-source models. Moreover, the execution-based metrics exhibit significant variability, confirming that performance optimization in a zero-shot setting is inherently unstable. Overall, this comparison suggests that, despite their general-purpose strength, closed-source models do not consistently outperform specialized fine-tuned models on the specific task of code efficiency optimization.

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

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

7. Conclusions and future developments

During this master's thesis we investigated automated code optimization through LLMs, focusing on efficiency improvement under strict functional correctness constraints. Unlike classical code generation tasks, optimization introduces additional structural and evaluative complexities: the mapping from inefficient to efficient code is inherently one-to-many, execution-based rewards are sparse and noisy, and performance measurements—especially in competitive programming settings—are often dominated by system-level variability. These characteristics make the task particularly challenging from both a modeling and an evaluation standpoint.
To address these difficulties, we proposed a structured two-stage fine-tuning framework that explicitly separates structural learning from execution-driven refinement. In the first stage, SFT was employed to expose the models to curated inefficient–efficient code pairs, enabling them to internalize recurring optimization patterns while preserving correctness. In the second stage, Reinforcement Learning Fine-Tuning (RLFT) based on Group Relative Policy Optimization (GRPO) incorporated execution feedback signals, with rewards gated by functional validity and modulated by relative runtime and memory variations. This sequential design was intended to stabilize training by preventing direct optimization on noisy execution signals from scratch.
The empirical results provide several important insights. SFT consistently improved structural alignment with efficient reference implementations, as measured by IOCCB, across both Qwen2.5-Coder and DeepSeek Coder families. This indicates that SFT effectively transfers recurring optimization patterns into the generation policy, going beyond superficial token-level edits. In most configurations, correctness either improved or remained stable, particularly for medium and large model scales, showing that structural specialization does not necessarily compromise semantic preservation. Moreover, memory usage exhibited coherent improvements after SFT, suggesting that the models learned to simplify data structures and reduce intermediate allocations. In con-

trast, relative speedup did not display a systematic positive trend. Given the very short execution times in the ACEOB benchmark, this behavior is likely attributable to measurement noise and limited runtime variability rather than to the absence of genuine optimization learning.
The RL stage revealed a substantially different dynamic. Although GRPO enabled stable convergence without requiring an explicit critic network, the reward signal was largely dominated by the correctness component. Once the models learned to consistently pass all test cases, reward growth rapidly saturated and execution-level improvements remained marginal. In some configurations, RLFT increased functional reliability but degraded structural alignment and memory efficiency. These findings suggest that, in environments where execution-based metrics are highly noisy and weakly informative, RL tends to reinforce safe and correct behaviors rather than drive consistent efficiency improvements. More broadly, execution-aware optimization requires sufficiently stable and discriminative reward signals; otherwise, policy-gradient methods converge toward conservative local optima.
A comparison with state-of-the-art proprietary models in a one-shot setting further contextualizes these findings. While closed-source systems demonstrated strong general-purpose capabilities and, in some cases, high correctness, they did not consistently achieve structural alignment with efficient references or reliable efficiency gains. In contrast, open-source models specialized through task-aware fine-tuning exhibited more coherent and interpretable behavior on the IC2EC task, showing that targeted adaptation can compete with much larger generalist systems in domain-specific optimization settings.
Several limitations remain. The competitive-programming nature of the benchmark constrains runtime variability, making fine-grained speed measurements intrinsically unstable. The reward formulation, although principled, remains relatively simple and correctness-dominated. Furthermore, the evaluation was conducted on a relatively small dataset and did not include large-scale industrial codebases or scientific computing workloads, where efficiency improvements may have greater practical impact.
Future developments should therefore focus on scaling and realism. Extending the framework to larger code-specialized models may enhance the ability to capture complex algorithmic transformations and long-range dependencies. At the same time, incorporating more diverse efficiency-oriented datasets—including real-world repositories and high-performance numerical applications—would provide stronger and more informative execution signals. Evaluating the approach in industrial environments, where runtime and memory usage directly affect cost and energy consumption, represents a crucial step toward practical deployment.
In conclusion, this thesis demonstrates that structural code optimization can be effectively learned through supervised specialization, while RL in execution-noisy environments remains sensitive to reward design and signal quality. Advancing toward larger models, richer datasets, and more realistic evaluation settings will be essential to transform LLMs from capable code generators into reliable, efficiency-aware co-optimizers that contribute to scalable and sustainable software development.

8. Bibliography and citations

References

[1] Abdul Ahad, Mir Sajjad Talpur, and Awais Jumani. Natural language processing challenges and issues: A literature review. *GAZI UNIVERSITY JOURNAL OF SCIENCE*, 36, 01 2023.

[2] Masood Anwar. Green computing and energy consumption issues in the modern age. *IOSR Journal of Computer Engineering*, 12:91–98, 01 2013.

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

[4] Paheli Bhattacharya, Manojit Chakraborty, Kartheek N S N Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, and Rishabh Gupta. Exploring large language models for code explanation, 2023.

[5] Peter F. Brown, Vincent J. Della Pietra, Peter V. de Souza, Jennifer C. Lai, and Robert L. Mercer. Class-based n-gram models of natural language. *Comput. Linguistics*, 18:467–479, 1992.

[6] Tom B. Brown, Benjamin Mann, and Ilya Sutskeveret al. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[7] Guendalina Caldarini, Sardar F. Jaf, and Kenneth McGarry. A literature survey of recent advances in chatbots. *CoRR*, abs/2201.06657, 2022.

[8] Mark Chen, Jerry Tworek, and Ilya Sutskever et al. Evaluating large language models trained on code, 2021.

[9] Aakanksha Chowdhery, Sharan Narang, and Jacob Devlin et al. Palm: Scaling language modeling with pathways, 2022.

[10] Google Cloud. Supervised fine-tuning for gemini llm. https://cloud.google.com/blog/products/ai-machine-learning/supervised-fine-tuning-for-gemini-llm.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[12] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. *CoRR*, abs/1905.03197, 2019.

[13] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents, 2025.

[14] Sanjay K. Dwivedi and Vaishali Singh. Research and reviews in question answering system. *Procedia Technology*, 10:417–424, 2013. First International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA) 2013.

[15] Jared Fernandez, Luca Wehrstedt, Leonid Shamis, Mostafa Elhoushi, Kalyan Saladi, Yonatan Bisk, Emma Strubell, and Jacob Kahn. Hardware scaling trends and diminishing returns in large-scale distributed training, 2025.

[16] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122, 2017.

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[18] Sabaat Haroon, Ahmad Faraz Khan, Ahmad Humayun, Waris Gill, Abdul Haddi Amjad, Ali R. Butt, Mohammad Taha Khan, and Muhammad Ali Gulzar. Assessing the impact of code changes on the fault localizability of large language models, 2026.

[19] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced BERT with disentangled attention. *CoRR*, abs/2006.03654, 2020.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997.

[21] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024.

[22] W. John Hutchins. The georgetown-ibm experiment demonstrated in january 1954. In Robert E. Frederking and Kathryn B. Taylor, editors, *Machine Translation: From Real Users to Research*, pages 102–114, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[23] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, January 2026.

[24] Jamin Rahman Jim, Md Apon Riaz Talukder, Partha Malakar, Md Mohsin Kabir, Kamruddin Nur, and M.F. Mridha. Recent advancements and challenges of nlp-based sentiment analysis: A state-of-the-art review. *Natural Language Processing Journal*, 6:100059, 2024.

[25] M I Jordan. Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. Technical report, California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science, 05 1986.

[26] Monisha Kanakaraj and Ram Mohana Reddy Guddeti. Nlp based sentiment analysis on twitter data using ensemble classifiers. In *2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN)*, pages 1–5, 2015.

[27] Denis Kocetkov, Raymond Li, and Loubna Ben Allal et al. The stack: 3 tb of permissively licensed source code, 2022.

[28] Nathan Lambert. Reinforcement learning from human feedback, 2026.

[29] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *CoRR*, abs/1901.07291, 2019.

[30] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019.

[31] 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.

[32] Yinhan Liu, Myle Ott, Naman Goyal, and et al. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[33] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 09 2011.

[34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[35] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

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

[37] Shushanta Pudasaini, Subarna Shakya, Sagar Lamichhane, Sajjan Adhikari, Aakash Tamang, and Sujan Adhikari. Application of nlp for information extraction from unstructured documents. In I. Jeena Jacob, Francisco M. Gonzalez-Longatt, Selvanayaki Kolandapalayam Shanmugam, and Ivan Izonin, editors, *Expert Clouds and Applications*, pages 681–690, Singapore, 2022. Springer Nature Singapore.

[38] L. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[39] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[40] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[42] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[43] Anna Rogers, Matt Gardner, and Isabelle Augenstein. Qa dataset explosion: A taxonomy of nlp resources for question answering and reading comprehension. *ACM Comput. Surv.*, 55(10), February 2023.

[44] Xin Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[46] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.

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

[48] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2024.

[49] Sonit Singh. Natural language processing for information extraction, 2018.

[50] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation, 2019.

[51] Jones Karen Sparck, Zampolli Antonio, Calzolari Nicoletta, and Palmer Martha. *Natural Language Processing: A Historical Review*, pages 3–16. Springer Netherlands, Dordrecht, 1994.

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

[53] Openai GPT-5 Team. Openai gpt-5 system card, 2025.

[54] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023.

[55] Hugo Touvron, Thibaut Lavril, Gautier Izacard, and Xavier Martinet et al. Llama: Open and efficient foundation language models, 2023.

[56] Hugo Touvron, Louis Martin, and Kevin Stone et al. Llama 2: Open foundation and fine-tuned chat models, 2023.

[57] A. M. TURING. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 10 1950.

[58] Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. Learn to code sustainably: An empirical study on llm-based green code generation, 2024.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[60] Analytics Vidhya. A brief overview of recurrent neural networks (rnn).

[61] IIIT Hyderabad Virtual Labs. N-grams and smoothing (theory).

[62] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*, 2023.

[63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[64] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9:36 – 45, 1966.

[65] Cameron R. Wolfe. Understanding and using supervised fine-tuning (sft) for language models. https://cameronrwolfe.substack.com/p/understanding-and-using-supervised.

[66] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.

A. Appendix A: Training Plots

In this appendix, we collect the complete set of training and validation curves monitored throughout the experiments, including metrics such as training loss, evaluation loss, and mean token accuracy. Additional diagnostic signals tracked during both SFT and RLFT are also reported.

A.1. SFT: Qwen Coder 2.5

As illustrated in Figures 8, 9, 10, 11, 12, and 13, all models exhibit early convergence. The evaluation loss stabilizes after a relatively small number of steps, and mean token accuracy quickly plateaus. At the same time, entropy steadily decreases, indicating progressive specialization of the policy and reduced output uncertainty. This behavior suggests that the models rapidly internalize the dominant structural patterns present in the inefficient–efficient pairs and that the available supervision signal is largely absorbed within the early training phase.

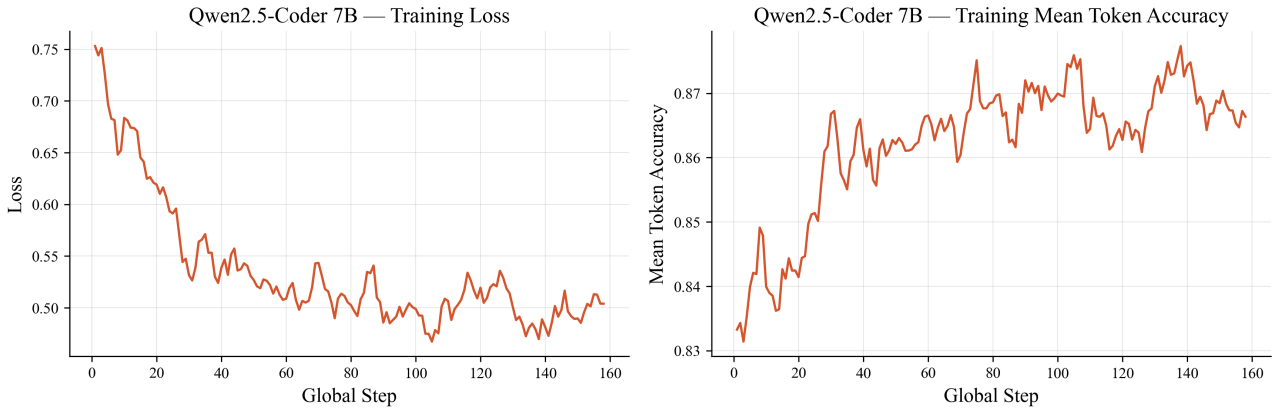


Figure 8: Train loss and train mean token accuracy for Qwen Coder 2.5 7b.

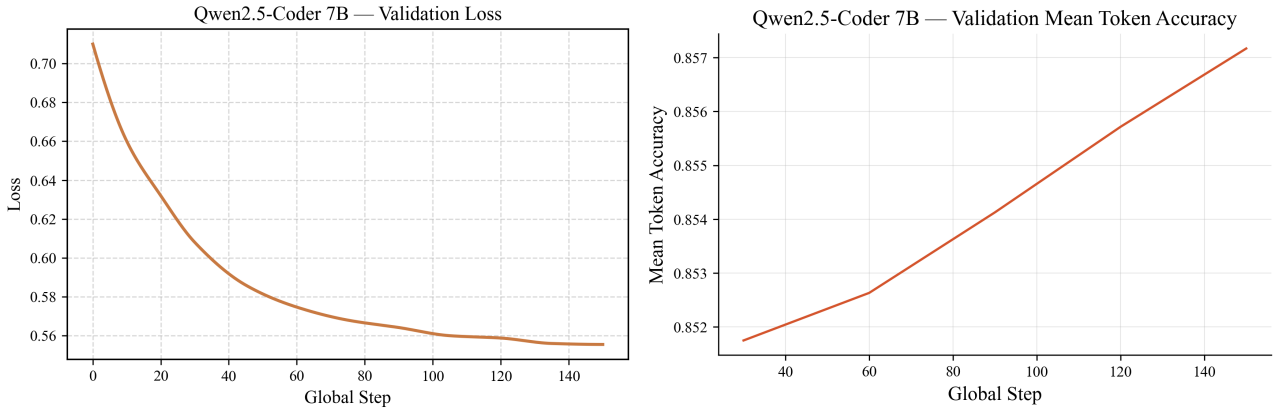


Figure 9: Eval loss and train mean token accuracy for Qwen Coder 2.5 7b.

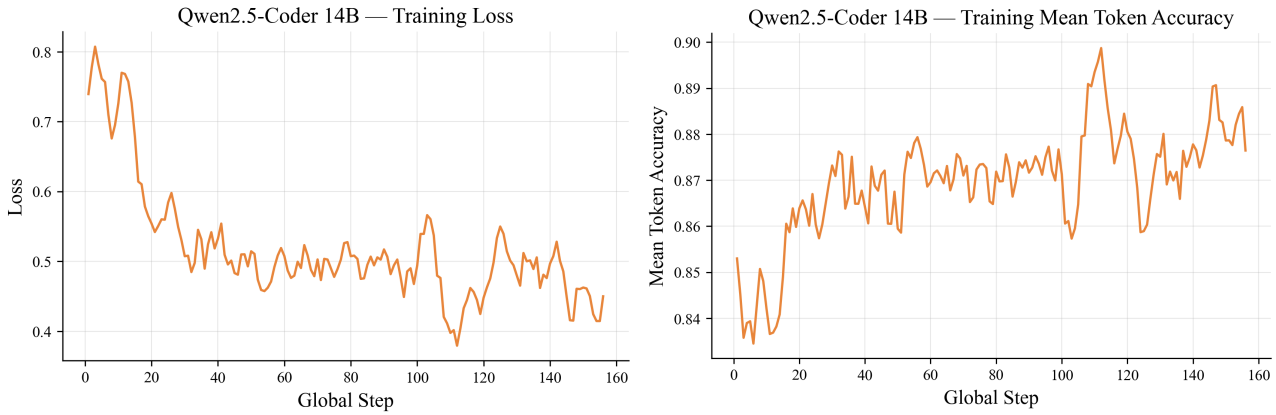


Figure 10: Train loss and train mean token accuracy for Qwen Coder 2.5 14b.

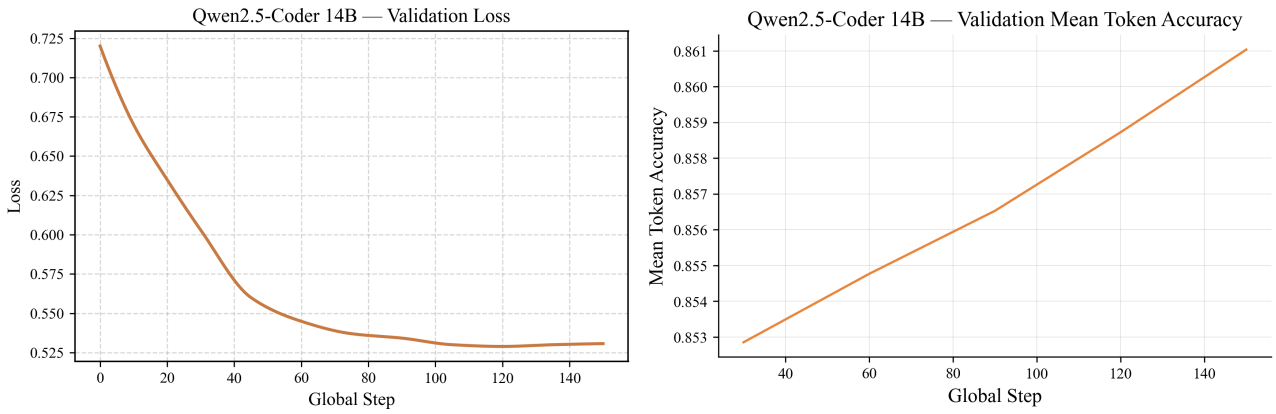


Figure 11: Eval loss and train mean token accuracy for Qwen Coder 2.5 14b.

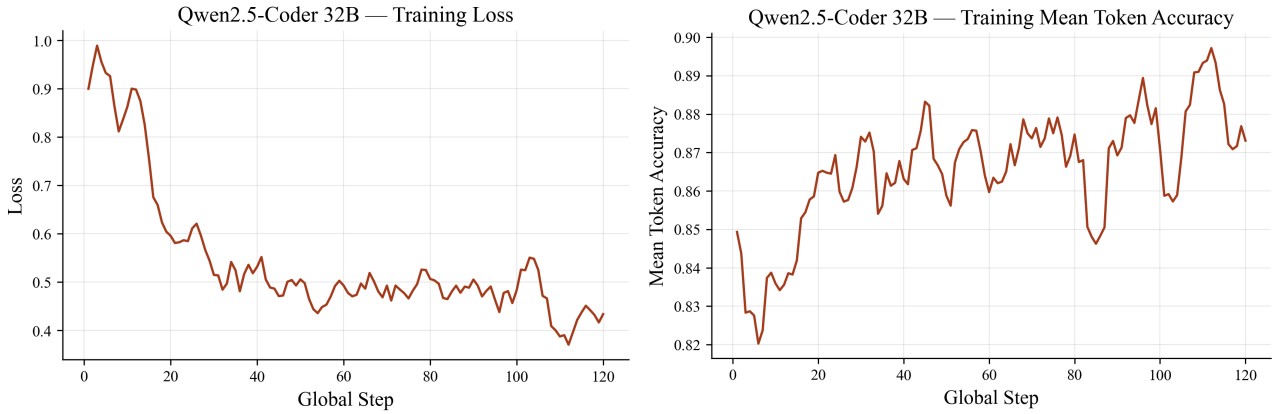


Figure 12: Train loss and train mean token accuracy for Qwen Coder 2.5 32b.

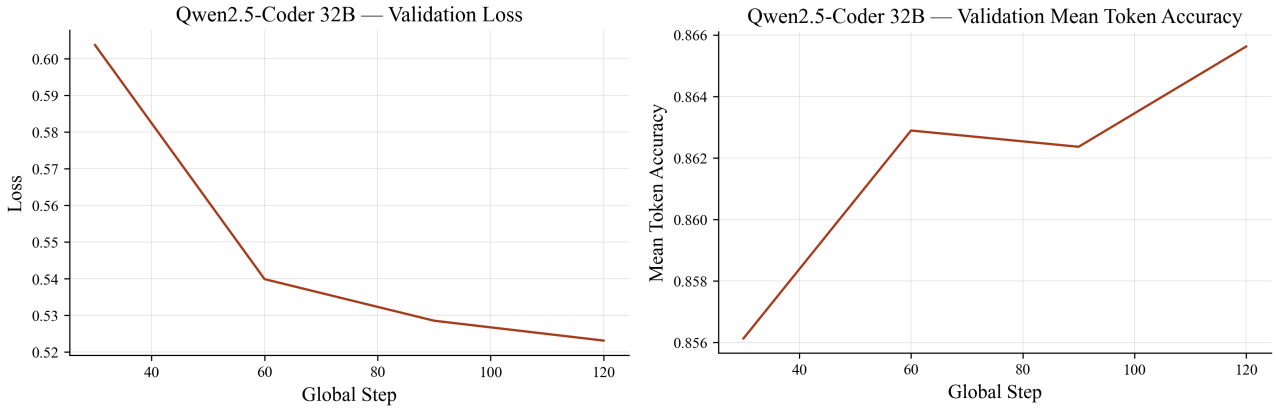


Figure 13: Eval loss and train mean token accuracy for Qwen Coder 2.5 32b.

A.2. SFT: Deepseek Coder

From a training perspective, the overall dynamics are consistent across scales, as shown in Figures 14, 15, 16, 17, 18, and 19, although convergence occurs over a longer horizon compared to the Qwen models. Evaluation loss gradually stabilizes between roughly 500 and 1000 steps, and mean token accuracy follows a similar trajectory, progressively plateauing as training advances. Entropy steadily decreases throughout training, indicating growing confidence and specialization toward the supervised objective. This slower but stable convergence suggests that DeepSeek models require more adaptation steps to fully internalize the optimization patterns present in the dataset, but ultimately reach a well-defined equilibrium.

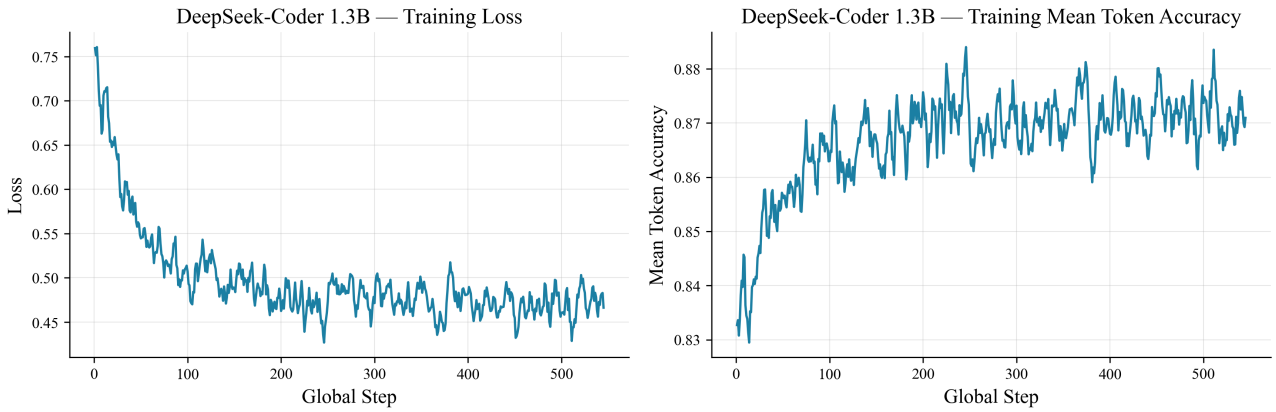


Figure 14: Train loss and train mean token accuracy for Deepseek Coder 1.3b.

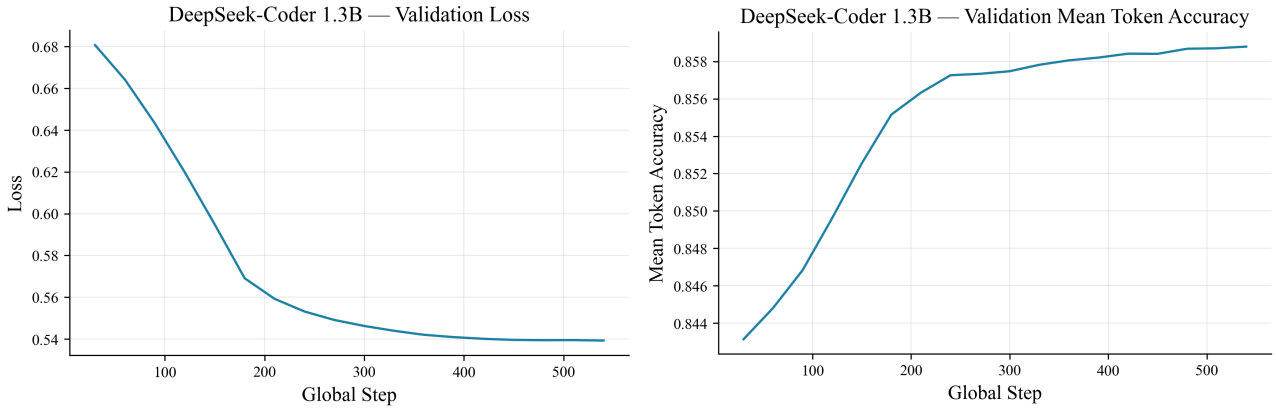


Figure 15: Eval loss and train mean token accuracy for Deepseek Coder 1.3b.

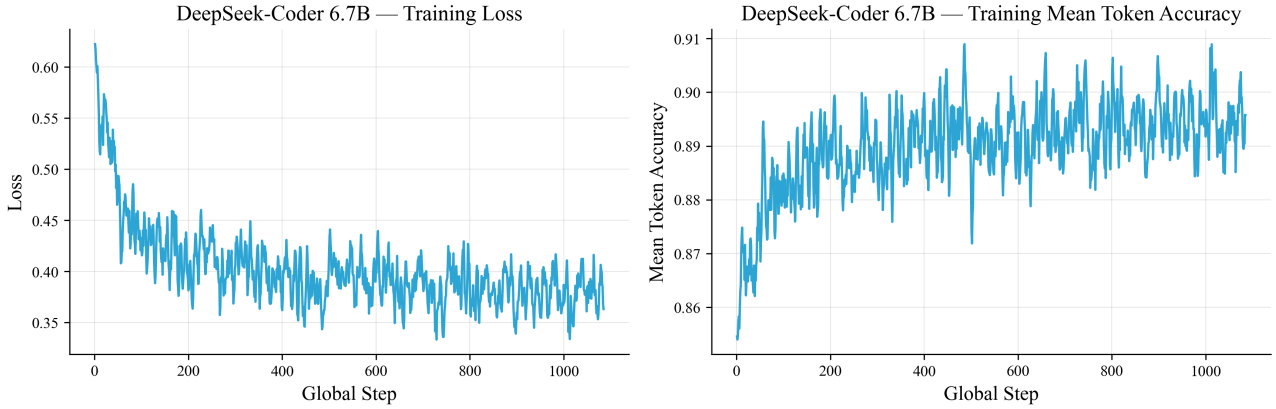


Figure 16: Train loss and train mean token accuracy for Deepseek Coder 6.7b.

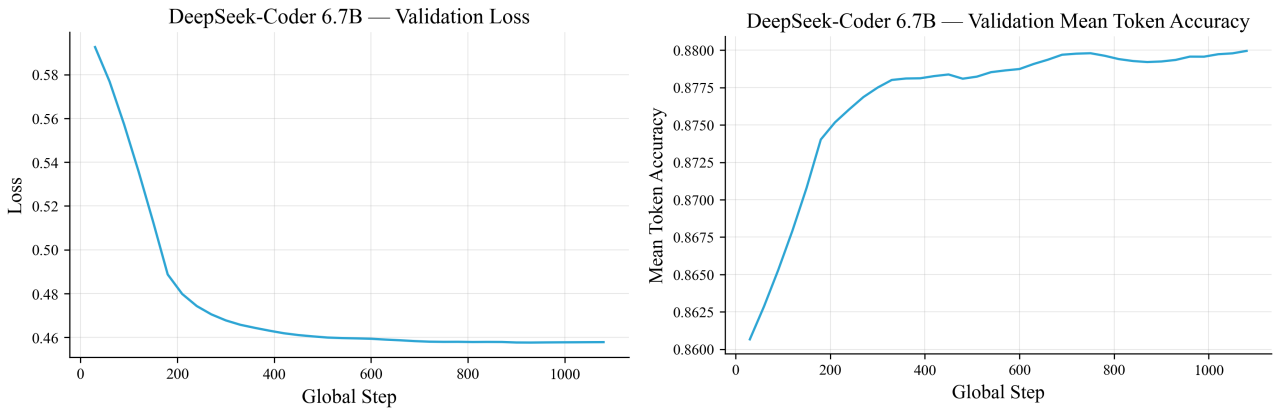


Figure 17: Eval loss and train mean token accuracy for Deepseek Coder 6.7b.

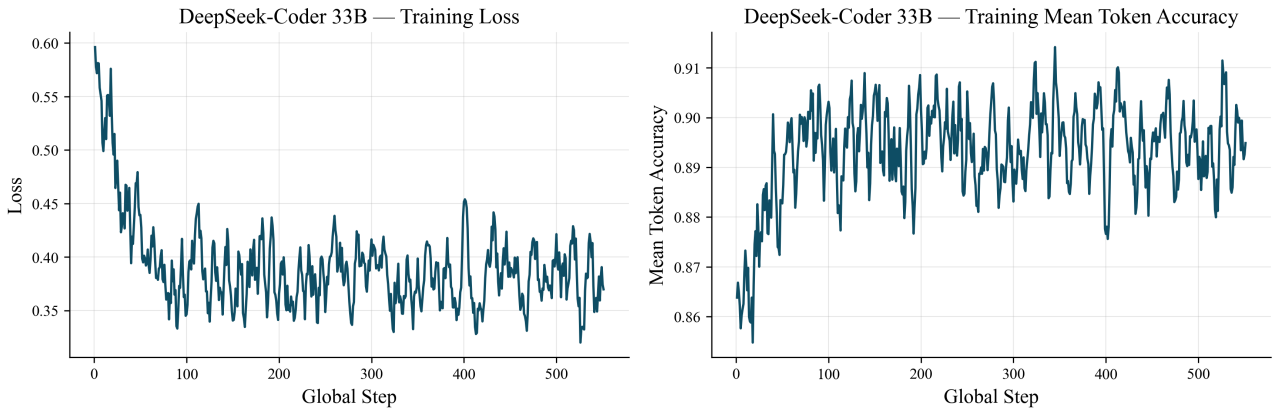


Figure 18: Train loss and train mean token accuracy for Deepseek Coder 33b.

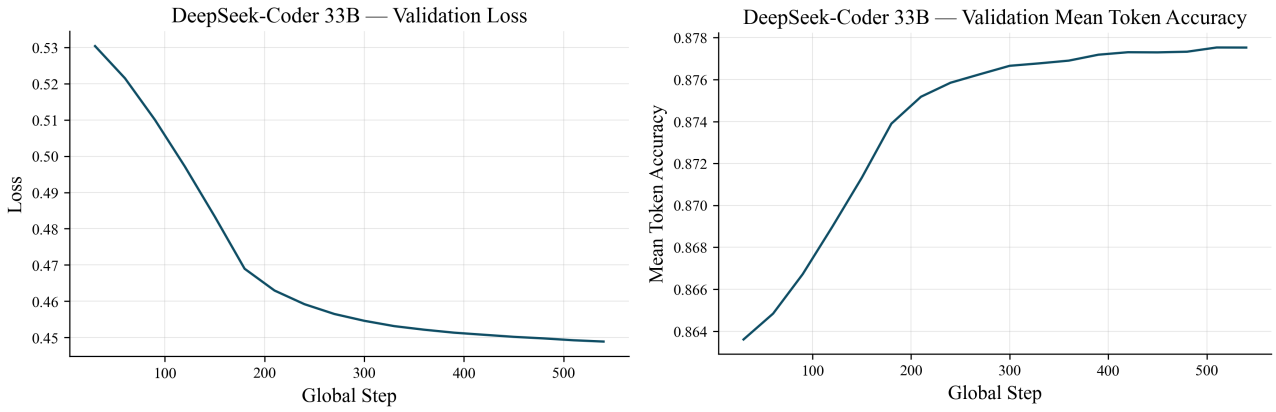


Figure 19: Eval loss and train mean token accuracy for Deepseek Coder 33b.

A.3. RLFT: DeepSeek Coder 1.3B

As illustrated in Figures 20, 21, and 22, both models rapidly increase their reward and then saturate around a value close to 1. Qwen reaches this plateau earlier, after roughly 200 steps, while DeepSeek requires around 500 steps, but the overall behavior is similar. The reward does not grow beyond this point because the dominant component of the signal is correctness. Once the model consistently generates solutions that pass the test cases, the reward effectively saturates. The timing and memory components are comparatively weak and highly noisy, so they do not provide a stable gradient signal capable of driving further improvement.
Entropy steadily decreases for both models, indicating reduced exploration and progressive stabilization of the policy. The loss oscillates around zero, which is consistent with a policy-gradient regime where advantages

are centered: once relative improvements disappear and rewards stabilize, updates fluctuate around equilibrium without clear directional progress. Overall, the training curves suggest convergence toward a correctness-focused local optimum rather than gradual refinement of execution efficiency.

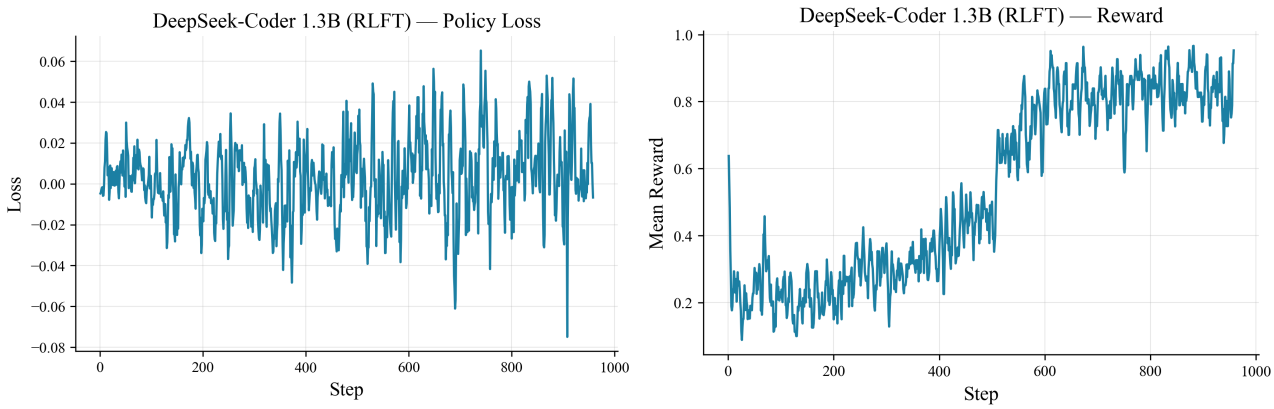


Figure 20: Training loss and reward during RLFT for DeepSeek Coder 1.3B.

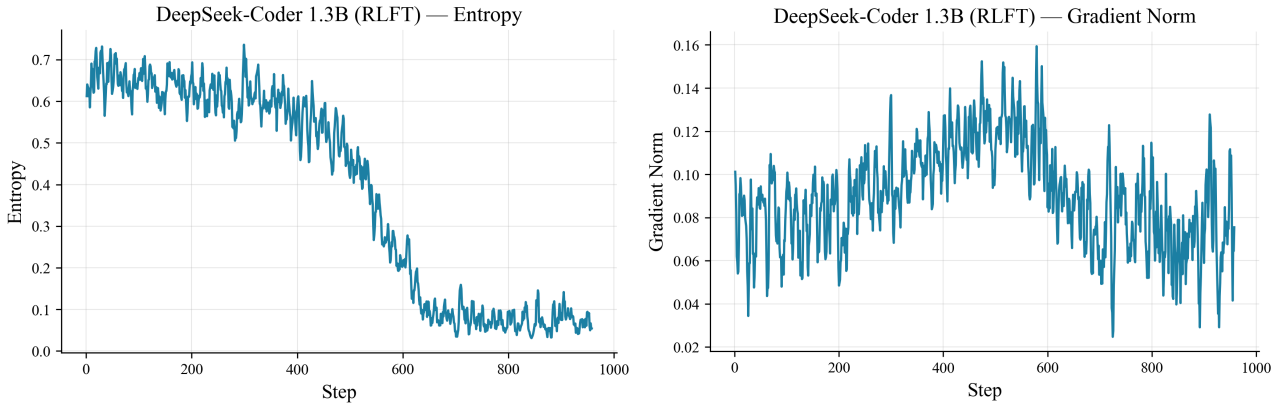


Figure 21: Policy entropy and gradient norm during RLFT for DeepSeek Coder 1.3B.

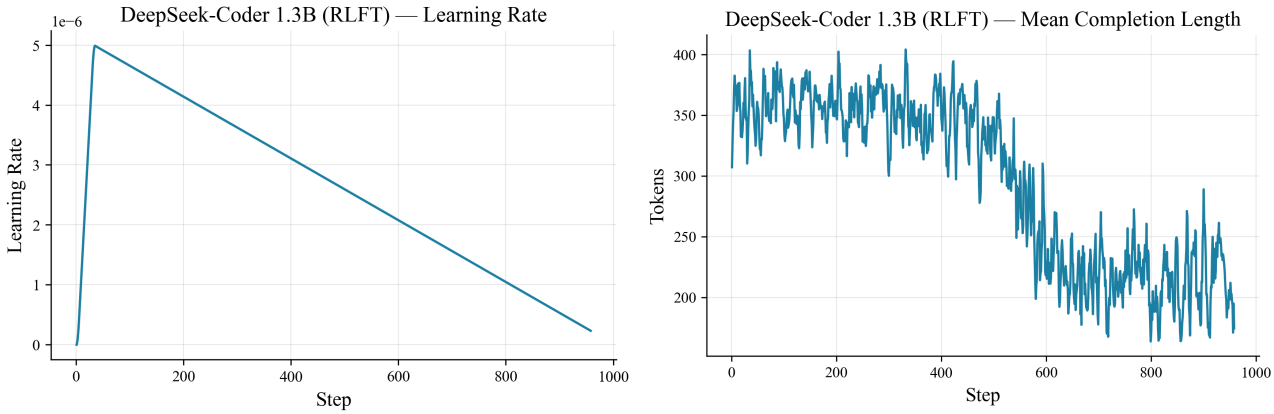


Figure 22: Learning rate schedule and mean completion length during RLFT for DeepSeek Coder 1.3B.

A.4. RLFT: Qwen Coder 2.5 7B

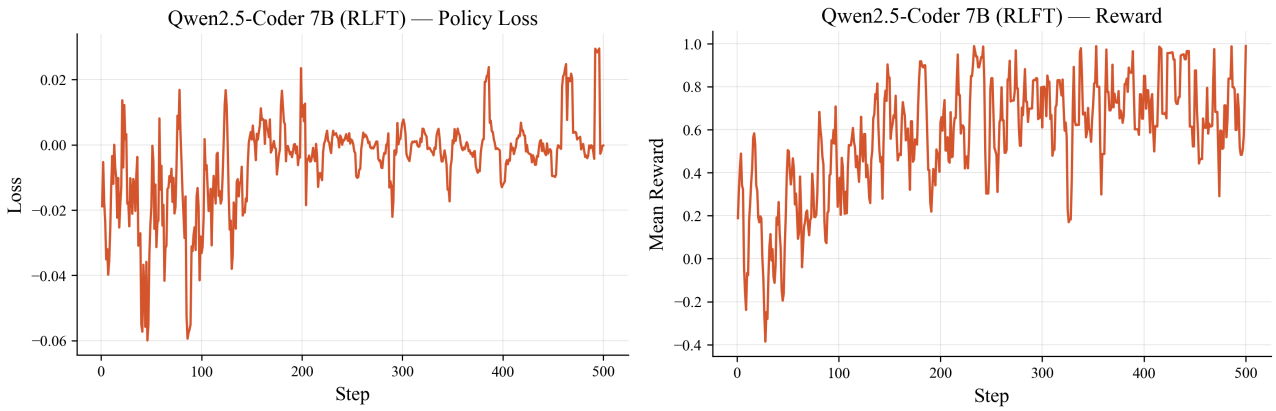


Figure 23: Training loss and reward during RLFT for Qwen Coder 2.5 7B.

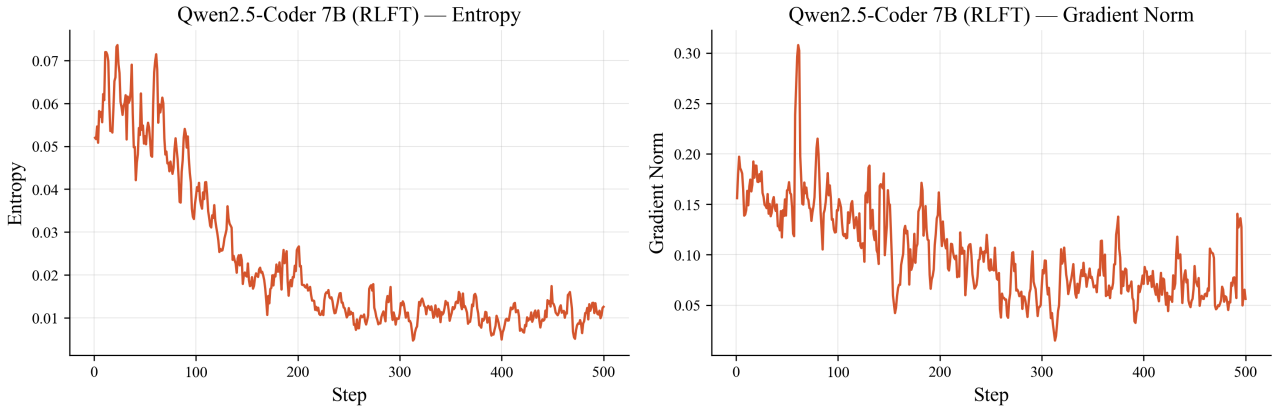


Figure 24: Policy entropy and gradient norm during RLFT for Qwen Coder 2.5 7B.

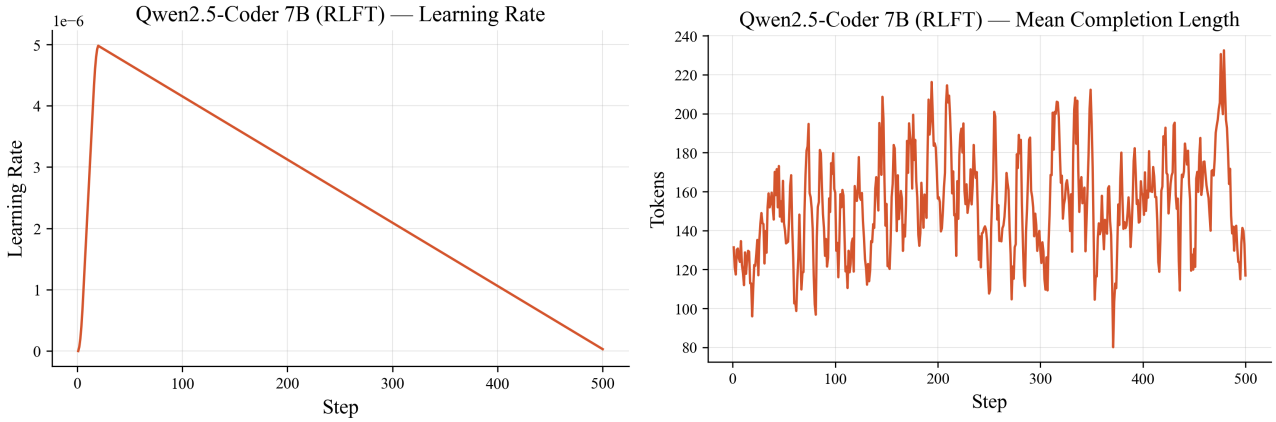


Figure 25: Learning rate schedule and mean completion length during RLFT for Qwen Coder 2.5 7B.

B. Appendix B: Run Configuration Tables

This appendix collects the full hyperparameter configuration used in the different training and evaluation runs.

Table 5: Training dataset configuration (shared across SFT runs).

Parameter	Value
Original dataset	ACEOB
Prompt format	Instruction chat format
Prompt template	`Optimize the following code for efficiency while keeping it functional correct. Use python constructs and libraries. Return only the optimized code, no extra text.\n{example['inefficient_code']}\n`
Input format	Unoptimized code
Target format	Optimized code
Total training examples	2444
Total test examples	100

Table 6: SFT configuration for Qwen Coder 2.5 models.

Parameter	Value
Model family	Qwen Coder 2.5 Instruct
Training framework	TRL — `SFTTrainer`
Objective	Completion-only cross-entropy loss
Optimizer	AdamW
Learning rate	2×10^{-5}
LR scheduler	Cosine decay
Number of epochs	3
Per-device train batch size	1
Gradient accumulation steps	16
Effective batch size	16
Mixed precision	BF16 enabled
FP16	Disabled
Evaluation strategy	Step-based
Evaluation	Every 20 steps
Eval subset size	50 samples

Table 7: SFT configuration for DeepSeek Coder models.

Parameter	Value
Model	DeepSeek Coder
Training framework	TRL — `SFTTrainer`
Objective	Completion-only cross-entropy loss
Optimizer	AdamW
Learning rate	2×10^{-5}
LR scheduler	Cosine decay
Number of epochs	3
Per-device train batch size	8
Gradient accumulation steps	2
Effective batch size	16
Mixed precision	BF16 enabled
FP16	Disabled
Evaluation strategy	Step-based
Evaluation	Every 20 steps
Eval subset size	80 samples

Table 8: RLFT configuration for Qwen2.5-Coder 7B.

Parameter	**Value**
Model	`AngeloCurti22/qwen_coder2.5-7b-sft` on HF
Dataset	`ACEOB top-1`
Sampling	8 generations
Max new tokens	1024
Temperature	0.7
top-p	0.95
Reward weights	correctness 1.0, speed 0.25, memory 0.05
Training epochs	1
Learning Rate	6.5×10^{-6}
batch size	8
grad. accum	2
max length	2048
Scheduler	warmup steps 15
LoRA	r=32; α=64; dropout 0.05
Precision / memory	BF16

Table 9: RLFT configuration for DeepSeek Coder 1.3B.

Parameter	**Value**
Model	`AngeloCurti22/deepseek-coder-1.3b-sft` on HF
Dataset	`ACEOB top-1`
Sampling	8 generations
Max new tokens	1024
Temperature	0.9
top-p	0.95
Reward weights	correctness 1.0, speed 0.25, memory 0.05
Training epochs	1
Learning Rate	5×10^{-6}
batch size	16
grad. accum	1
max length	2048
Scheduler	warmup steps 30
LoRA	r=16; α=32; dropout 0.05
Precision / memory	BF16

Table 10: Test-set evaluation configuration (shared across SFT models).

Parameter	**Value**
Dataset	ACEOB top-1 test
Number of test samples	100
Prompt format	Instruction chat format
Prompt template	`Optimize the following code for efficiency while keeping it functional correct. Use python constructs and libraries. Return only the optimized code, no extra text.\n{example['inefficient_code']}\n`
Input format	Unoptimzied code
Target format	Optimized code
Evaluation mode	Autoregressive generation
Decoding strategy	Autoregressive stochastic sampling
Max new tokens	512
Temperature	0.2
Sampling	Enabled (`do_sample=True`)
Top-p / Top-k	Not used
Execution device	Single GPU

Abstract in lingua italiana

I progressi dell'hardware non sono più sufficienti a compensare inefficienze software, rendendo le prestazioni un fattore centrale nell'aumento dei costi operativi e dei consumi energetici. Questa dinamica è particolarmente evidente in Eni S.p.A., le cui attività si fondano su simulazioni numeriche su larga scala, dove l'efficienza computazionale incide direttamente su costi, tempi di esecuzione e impatto energetico. Ciò motiva la ricerca sull'ottimizzazione automatica del codice mediante Large Language Models (LLM).
Nonostante tale rilevanza, l'ottimizzazione orientata alle prestazioni resta meno esplorata rispetto ai task di generazione e comprensione del codice.
Questo lavoro indaga se code-LLM possano essere adattati sistematicamente all'ottimizzazione del codice. Il contributo principale consiste in un framework a due stadi che separa l'apprendimento strutturale dal raffinamento guidato dall'esecuzione. In una prima fase, un puro Supervised Fine-Tuning (SFT) su coppie inefficiente–efficiente consente di interiorizzare pattern di trasformazione ricorrenti. Successivamente, il modello è adattato tramite Reinforcement Learning Fine-Tuning (RLFT), basato su Group Relative Policy Optimization (GRPO), che stabilizza l'addestramento mediante confronti relativi tra soluzioni candidate e allinea la generazione del modello a segnali derivanti dall'esecuzione del codice.
Il framework è valutato sui modelli Qwen2.5-Coder e DeepSeek-Coder, a diverse scale parametriche, utilizzando il benchmark ACEOB. Le prestazioni sono misurate in termini di allineamento strutturale, correttezza funzionale ed efficienza relativa. I risultati mostrano che lo SFT migliora in modo consistente allineamento strutturale ed efficienza in memoria, con incrementi di correttezza alle scale medio-grandi. L'RLFT rafforza principalmente l'affidabilità funzionale, ma non produce miglioramenti stabili sul runtime, evidenziando i limiti dei segnali esecutivi rumorosi.
Nel complesso, emerge che i pattern di ottimizzazione possono essere appresi efficacemente tramite supervisione, mentre l'allineamento all'efficienza mediante RL rimane fortemente dipendente dalla qualità del segnale di reward.

Parole chiave: LLM, Ottimizzazione del Codice, Reinforcement Learning, Fine Tuning, Group Relative Policy Optimization