

Beyond Correctness: Benchmarking LLMs for Algorithmic Program Optimization

Yehua Lin¹, Yin Chen^{2,*}

¹School of Computer Science, South China Normal University, China

²School of Artificial Intelligence, South China Normal University, China

linyehua@m.scnu.edu.cn ychen@scnu.edu.cn

Abstract

Large language models (LLMs) can generate functionally correct code, but their ability to optimize programs is underexplored. We introduce OEISPerf, a code-to-code benchmark for algorithmic program optimization built from alternative implementations of integer sequences in the On-Line Encyclopedia of Integer Sequences. For each sequence we mine multiple mathematical formulations, synthesize and validate Python implementations, and pair a substantially slower program with the fastest validated reference found by our construction pipeline, producing instances that often require substantial structural transformations and can yield substantial speedups. Evaluating 15 open-source and closed-source LLMs shows that even the strongest evaluated setting, Gemini-3-Flash with its thinking mode enabled, matches the reference efficiency on only 67.9% of test instances. Across LLMs, we observe both incorrect “optimizations” that break correctness and superficial refactorings that preserve suboptimal complexity. Finally, we fine-tune Qwen2.5-Coder-3B-Instruct via reinforcement learning, markedly improving reference attainment on OEISPerf while slightly improving performance on standard code-generation benchmarks.¹

1 Introduction

Large language models (LLMs) have achieved remarkable progress on a wide range of programming tasks (Jiang et al., 2026; Huynh and Lin, 2025). However, their effectiveness in program performance optimization is still not well understood. Program performance optimization is important and challenging because it requires transforming an inefficient yet functionally correct implementa-

tion into a substantially faster one while preserving its semantics (Garg et al., 2022).

Recent work such as PIE (Shypula et al., 2024) has taken an important step in this direction by constructing code optimization datasets from human submission logs and using them to evaluate the optimization capability of LLMs. Despite these contributions, such datasets may not fully reflect the type of optimization behavior that we aim to study. Human contestants in programming competitions typically begin from a submission that receives a Time Limit Exceeded verdict and iteratively refine their code until it just satisfies the time limit. Once the verdict becomes Accepted they often stop further optimization. Consequently, the collected logs primarily capture small, incremental improvements on code that is already near the time limit. Typical edits include replacing C++ input and output streams `cin` and `cout` with `scanf` and `printf`, or switching from dynamic arrays such as `vector` to static arrays. These modifications often result in relatively modest speedups, and such engineering-level optimizations do not adequately assess the inductive reasoning ability of LLMs for program optimization.

To address these issues, we propose OEISPerf. OEISPerf does not rely on mined submission logs. It instead leverages the On-Line Encyclopedia of Integer Sequences (OEIS), which is one of the most comprehensive repositories of integer sequences (Sloane, 2023). Many sequences in OEIS admit multiple implementation strategies. For a single sequence, there may exist an exponential time recursive procedure as well as an efficient closed-form computation. We mine these alternative strategies from OEIS, use LLMs to convert them into Python programs, and validate the generated code using the known terms of each sequence. For every sequence, we select the fastest validated program discovered by our pipeline as the efficient reference solution, which serves as an empirical optimization

*Corresponding author.

¹The code and dataset are publicly available at <https://github.com/l-yh0/OEISPerf> and <https://huggingface.co/datasets/l-yh/OEISPerf>.

target. We then remove programs whose performance is too close to this reference. The remaining programs for the same sequence are treated as inefficient solutions and are paired with the efficient reference solution to form optimization instances.

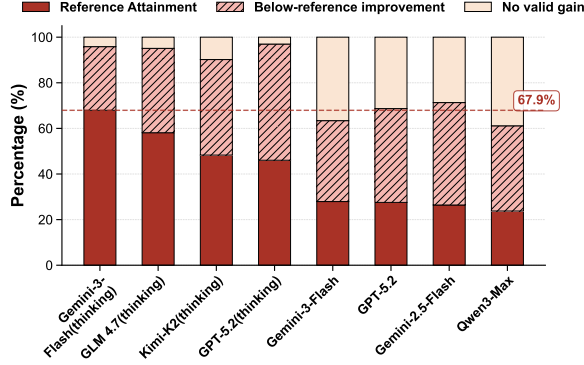


Figure 1: Performance distribution of selected representative LLMs on OEISPerf, decomposed into three mutually exclusive outcomes: (1) **Reference attainment**, where the generated program is functionally correct and achieves an execution time no larger than that of the efficient reference implementation, $t(p) \leq t(p_{\text{ref}})$; (2) **Below-reference improvement**, where the generated program is functionally correct and no slower than the inefficient implementation, $t(p) \leq t(p_{\text{ineff}})$, but still slower than the reference; and (3) **No valid gain**, including cases with performance degradation, $t(p) > t(p_{\text{ineff}})$, execution failures, or functional incorrectness. The suffix “(thinking)” indicates that the model is evaluated with its thinking mode enabled; models without this suffix are evaluated with the mode disabled or when such an option is not available. The horizontal dashed line marks the 67.9% reference attainment rate achieved by Gemini-3-Flash (thinking).

We evaluate 15 LLMs on OEISPerf. Figure 1 reports results for several representative models. Even the strongest evaluated setting, Gemini-3-Flash (thinking) optimizes only 67.9% of the programs to match the efficient reference solution, suggesting that OEISPerf poses a substantial challenge. To further explore the potential of OEISPerf, we perform reinforcement learning fine-tuning on Qwen2.5-Coder-3B-Instruct using the training split of OEISPerf and observe considerable performance improvements.

In summary, our main contributions are as follows:

- We introduce OEISPerf, a benchmark for algorithmic program optimization built from alternative implementations of OEIS integer sequences.

- We evaluate 15 LLMs on OEISPerf, showing that current models often either break correctness during aggressive rewrites or remain trapped in suboptimal algorithmic structures.
- We show that GRPO fine-tuning on the OEISPerf training split substantially improves the program optimization ability of Qwen2.5-Coder-3B-Instruct, while slightly improving its performance on standard code-generation benchmarks.

2 OEISPerf Dataset

2.1 Data Construction

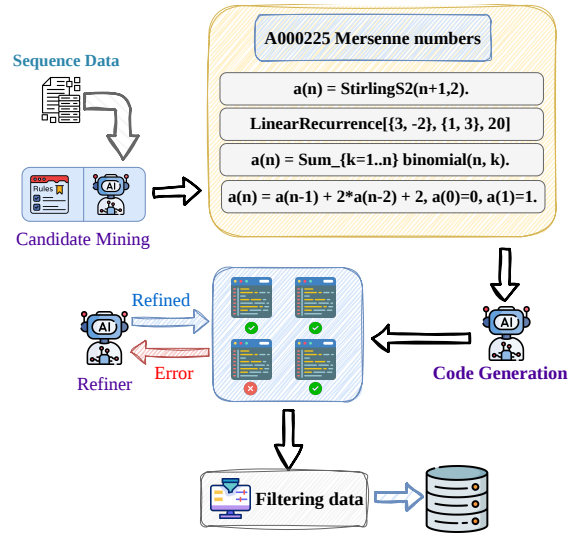


Figure 2: Overview of the OEISPerf construction pipeline.

Step 0: Sequence data preprocessing. We first collect sequence data from OEIS, and then use Qwen3-14B to extract, from the OEIS entries, the alternative computation specifications associated with each sequence, such as recurrences, closed-form expressions, and generating functions.

Step 1: Code generation. We use DeepSeek-V3.2 to generate sequence computation programs according to the specifications extracted in Step 0. In this step, we instruct the model to avoid additional mathematical reasoning or rewrites aimed at improving asymptotic complexity, and instead to implement the provided specifications as faithfully as possible.

Step 2: Program validation. For each candidate program of a sequence, we validate it against all the terms extracted from OEIS by checking whether

the program outputs match the given sequence. We also examine whether the program hard-codes the full set of terms used for validation, and we exclude such programs from subsequent evaluation.

Step 3: Program repair. For programs in Step 2 that fail to execute or do not pass validation, we provide the runtime error messages, the corresponding source code, and the original specification to the LLM in order to obtain a repaired version.

Step 4: Runtime evaluation. We evaluate the runtime on the last $k = 7$ terms provided by the OEIS entry. For sequences with fewer than k available terms, we use all available terms. For these last k terms, the actual input parameter n is typically large enough to expose significant performance differences. To reduce noise from hardware and system variability, we time m iterations over all k terms, repeat for r rounds, and use the median across rounds to compute the average per-call runtime.

Step 5: Constructing program pairs. For each sequence, we select the program with the smallest runtime as the efficient reference solution p_{ref} . Among the remaining candidates, we discard programs whose runtime lies within a factor of the base of the natural logarithm e of that of p_{ref} , in order to obtain higher-quality program pairs. Each remaining program is then treated as an inefficient program p_{ineff} , from which we remove all comments in the source code, and paired with p_{ref} for subsequent experiments.

2.2 Dataset Splitting

To mitigate data leakage caused by near-duplicate programs or shared mathematical structures, we split the dataset by sequence identifier. All instances derived from the same sequence are assigned to the same split. We utilize a random partition for training, validation, and testing, which results in 2762 training instances, 240 validation instances, and 265 test instances.

2.3 Test Set Data Analysis

Tree Edit Distance Algorithm-level program optimization may lead to substantial restructuring of the control flow. To objectively quantify this structural change, we apply the Zhang–Shasha algorithm (Zhang and Shasha, 1989) to the Abstract Syntax Trees (ASTs) of our program pairs to compute their tree edit distance.

Metric	Edit Distance	Norm. Dist.
Mean	66.3	0.74
Median	54.0	0.75
Min / Max	11 / 227	0.45 / 0.94
Threshold	% of Pairs Exceeding	
> 20	97.4%	
> 50	53.6%	
> 100	17.7%	

Table 1: AST Tree Edit Distance statistics between p_{ineff} and p_{ref} on the test split of OEISPerf. “Norm. Dist.” denotes the edit distance divided by the maximum AST node count of each pair.

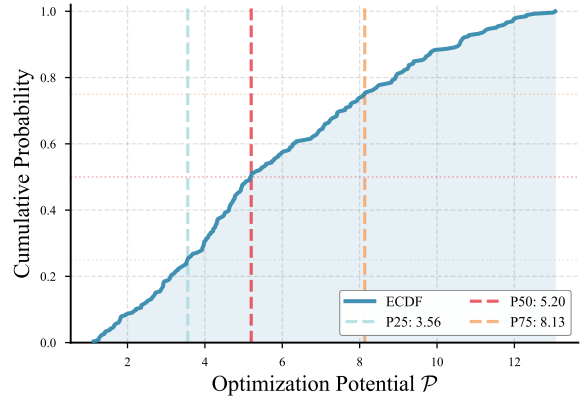


Figure 3: **Empirical cumulative distribution function (ECDF) of optimization potential.** Distribution of the optimization potential $\mathcal{P} = \ln t(p_{ineff}) - \ln t(p_{ref})$. Dashed vertical lines delineate the 25th, 50th, and 75th percentiles (P_{25} , P_{50} , and P_{75}) of the distribution.

The results in Table 1 indicate that structural differences between p_{ineff} and p_{ref} are substantial for most instances, with normalized distances concentrated around 0.75 and typical edit counts requiring modification of a large fraction of the AST. Taken together, these results indicate that OEISPerf often goes beyond simple cosmetic refactoring and frequently requires meaningful structural reorganization to match the efficiency of the reference implementations.

Optimization Potential Distribution To further characterize the difficulty of OEISPerf, we analyze the optimization potential available for each test instance. We define the optimization potential $\mathcal{P}$ as $\mathcal{P} = \ln t(p_{ineff}) - \ln t(p_{ref})$.

Figure 3 shows the empirical cumulative distribution function (ECDF) of this optimization potential on the test split. The distribution is strongly right skewed: the 25th, 50th, and 75th percentiles are $P_{25} = 3.56$, $P_{50} = 5.20$, and $P_{75} = 8.13$, which correspond to typical speedups from roughly tens

Model	Params	Reference attainment rate (%)	Reference-normalized speedup score	No valid gain rate (%)
<i>Open-Source Models</i>				
GLM 4.7 (thinking)	—	58.1	91.4	4.9
Kimi-K2 (thinking)	1T (32B)	48.3	83.8	9.8
Qwen3-14B (thinking)	14.8B	37.4	78.3	14.3
GPT-OSS-20B (thinking)	21B (3.6B)	37.7	78.6	11.3
DeepSeek-V3.2 (thinking)	685B	33.2	71.6	18.5
DeepSeek-V3.2	685B	17.4	49.7	39.6
GLM 4.7	—	16.2	43.8	49.4
Qwen3-14B	14.8B	10.9	38.1	42.6
Llama-3.1-8B-Instruct	8B	3.4	18.4	71.7
Qwen2.5-Coder-3B-Instruct	3B	2.3	21.6	70.2
Llama-3.2-3B-Instruct	3B	1.9	13.2	79.2
<i>Closed-Source Models</i>				
Gemini-3-Flash (thinking)	—	67.9	93.7	4.2
GPT-5.2 (thinking)	—	<u>46.0</u>	<u>89.2</u>	3.0
Seed-1.8 (thinking)	—	37.4	86.8	<u>3.8</u>
Gemini-2.5-Flash (thinking)	—	30.9	70.1	21.5
Gemini-3-Flash	—	27.9	59.6	36.6
GPT-5.2	—	27.5	60.7	31.3
Gemini-2.5-Flash	—	26.4	64.6	28.7
Qwen3-Max	—	23.8	54.2	38.9
Claude-Sonnet-4.5	—	21.1	59.0	29.8
Claude-Haiku-4.5	—	18.9	56.6	27.9
Seed-1.8	—	10.6	33.1	62.3

Table 2: Performance comparison of different models. For MoE models, total parameters are shown with activated parameters in parentheses. The suffix (*thinking*) indicates that the model was evaluated with its thinking mode enabled, whereas models without this suffix were evaluated with the mode disabled or with no such option available. Bold denotes the best result and underlining denotes the second-best result within each metric group.

to well over a thousand times faster than the inefficient baseline. This indicates that OEISPerf does not focus on micro-optimizations, but instead probes the LLMs’ ability to realize substantial algorithmic improvements in program efficiency.

2.4 Metrics

We consider three primary metrics: reference attainment rate, reference-normalized speedup score, and no valid gain rate.

Reference attainment rate. We define the reference attainment rate as the proportion of instances for which the generated program is functionally correct and its runtime is no greater than that of the efficient reference program p_{ref} . This metric is used to assess the extent to which the model attains reference-level performance for code optimization on the given task.

Reference-normalized speedup score. We define the reference-normalized speedup score as

$$S = \frac{100}{|\mathcal{T}|} \sum_{p \in \mathcal{T}} \text{clip} \left(\frac{\ln t(p_{ineff}) - \ln t(p)}{\ln t(p_{ineff}) - \ln t(p_{ref})}, 0, 1 \right), \quad (1)$$

Here, $t(\cdot)$ denotes the runtime of a program, $\mathcal{T}$ denotes the evaluation set, and $\text{clip}(\cdot, 0, 1)$ truncates the value to the interval $[0, 1]$. For an incorrect program, the corresponding term in the summation is defined to be zero. This metric characterizes, on a common scale, the speedup achieved by the generated program relative to the inefficient baseline and the efficient reference.

No valid gain rate. We define the no valid gain rate as the proportion of instances in which the LLM output is invalid or slower than the initial inefficient program. Concretely, an instance is counted as no valid gain if the generated program does not pass validity checks, or if its runtime exceeds that of the initial inefficient program p_{ineff} .

3 Evaluation on OEISPerf

3.1 Experimental Setup

LLMs. We evaluated OEISPerf on a total of 15 LLMs. Open-source models include Kimi-K2 (Kimi Team, 2025), GLM 4.7 (GLM, 2025), GPT-OSS-20B (OpenAI, 2025a), Qwen3-14B (Qwen Team, 2025), DeepSeek-V3.2 (DeepSeek-AI, 2025), Qwen2.5-Coder-3B-Instruct (Hui et al., 2024), Llama-3.2-3B-

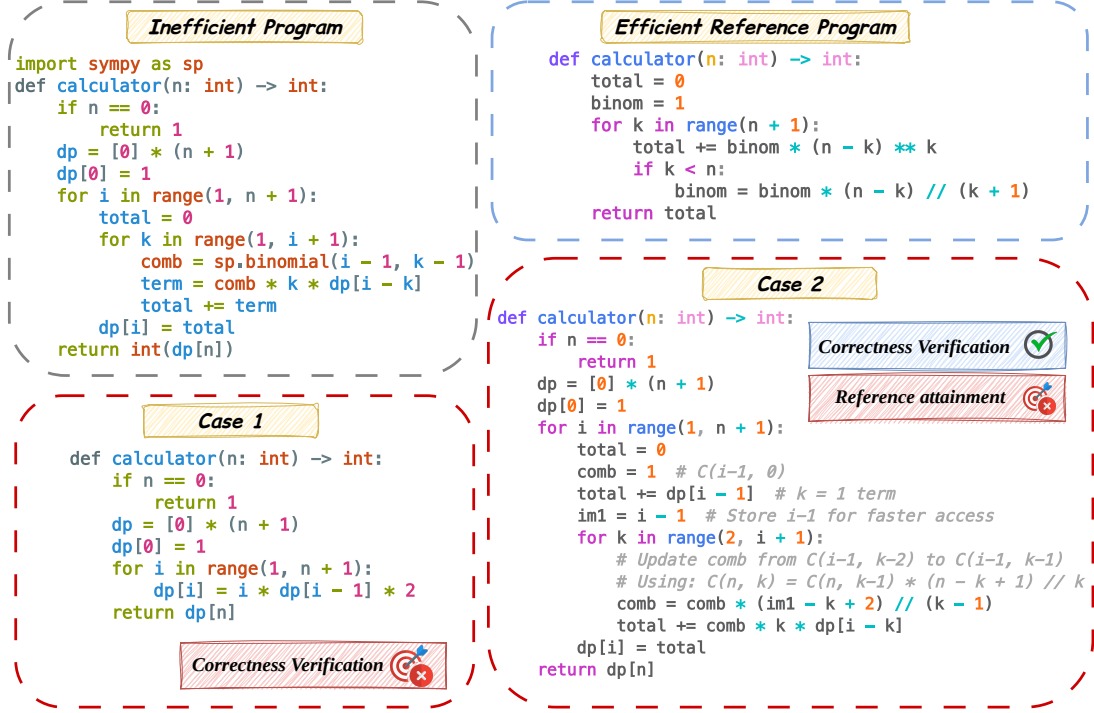


Figure 4: Case study of optimization solutions generated by LLMs for OEIS sequence A000248. The upper panel presents a representative dataset instance by contrasting the inefficient program with the efficient reference program. The inefficient program implements the recursive definition $a(n) = \sum_{k=1}^n \binom{n-1}{k-1} k a(n-k)$ using a standard dynamic programming approach. The efficient reference uses a single-pass summation by evaluating the explicit summation $a(n) = \sum_{k=0}^n \binom{n}{k} (n-k)^k$ and iteratively updating the binomial coefficient. The lower panel illustrates two prevalent failure modes encountered by LLMs during the optimization process. Case 1 displays an invalid complexity reduction where the model attempts an $O(n)$ solution but introduces logical errors that violate functional correctness. Case 2 exemplifies a partial optimization where the model successfully produces a functionally correct program that slightly streamlines the inner combinatorial calculation. However, this specific output preserves the suboptimal nested-loop structure and strictly bounds the overall time complexity to $O(n^2)$. Such partial improvements fail to bridge the fundamental algorithmic gap required to match the reference performance.

Instruct (Llama Team, 2024), and Llama-3.1-8B-Instruct (Llama Team, 2024). Closed-source models include Gemini-3-Flash (Google DeepMind, 2025), Gemini-2.5-Flash (Gemini Team, 2025), Seed-1.8 (ByteDance Seed, 2025), GPT-5.2 (OpenAI, 2025b), Qwen3-Max (Qwen Team, 2025), Claude Sonnet 4.5 (Anthropic, 2025b), and Claude Haiku 4.5 (Anthropic, 2025a). For each task instance, we sample exactly one solution. We use the temperature 1.0, and maximum output length of 32,768 tokens across model settings.

Prompt. The LLMs are provided only with the inefficient Python program through the prompt template shown in Appendix Table 12, with comments and explicit metadata removed where possible.

3.2 Main Results

Table 2 presents the evaluation results of LLMs on the test split of OEISPerf. Even the strongest evaluated setting, Gemini-3-Flash with thinking mode

enabled, matches the efficiency of the efficient reference solution on only 67.9% of test instances. The open-source model GLM-4.7 (thinking) comes close to this level, achieving reference attainment rates above one half, together with high reference-normalized speedup score and single-digit no valid gain rates.

Interestingly, Seed-1.8 (thinking) obtains a low no valid gain rate and a high reference-normalized speedup score, but its reference attainment rate is much lower than that of other top-performing LLMs. This suggests that some LLMs have learned important optimization strategies and can often substantially accelerate inefficient programs, yet still find it difficult to match the efficient reference solutions. In other words, these models exhibit strong optimization capabilities, but this capability is still insufficient to push inefficient programs to a fully optimized level.

We observe that LLMs exhibit a pronounced

dependence on the thinking mode. For Gemini-3-Flash, disabling thinking mode reduces the reference attainment rate by more than half and increases the no valid gain rate, and several other LLMs show similar trends. This phenomenon indicates that OEISPerf requires LLMs to perform deep inductive reasoning to derive strong algorithms for improving inefficient programs, rather than merely applying shallow, local code edits.

3.3 Case Study

Figure 4 illustrates two common optimization failure modes. First, the model may attempt an aggressive algorithmic rewrite to reach the reference efficiency, but due to insufficient inductive reasoning it fails to preserve semantic equivalence, causing the optimized code to fail correctness checks (Case 1). Second, the model may preserve functional correctness but only perform local refactoring within the original control-flow structure, resulting in performance that still falls short of the reference solution (Case 2). Overall, these cases suggest that current LLMs still have substantial room for improvement in program optimization tasks.

4 Reinforcement Learning

4.1 Group Relative Policy Optimization

To investigate the enhancement of code optimization capabilities in LLMs, we conduct reinforcement learning fine-tuning on our constructed training set. We employ Group Relative Policy Optimization (GRPO) as the designated training algorithm (Shao et al., 2024), with training implemented based on the verl framework (Sheng et al., 2025). GRPO effectively mitigates the high variance inherent in code generation tasks and eliminates the need for a separate value network, thereby reducing computational overhead.

Formally, for each query q , GRPO samples a group of outputs $\{o_1, o_2, \dots, o_G\}$ from the old policy $\pi_{\theta_{\text{old}}}$. Consequently, the loss is defined as follows:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\substack{q \sim P(Q), \\ \{o_i\} \sim \pi_{\theta_{\text{old}}}}} \left[\frac{1}{G} \sum_{i=1}^G \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \left(\min \left(\rho_{i,t} \hat{A}_{i,t}, \right. \right. \right. \\ \left. \left. \left. \text{clip}(\rho_{i,t}, 1 - \epsilon, 1 + \epsilon) \hat{A}_{i,t} \right) - \beta \mathbb{D}_{KL}(\pi_{\theta} \parallel \pi_{\text{ref}}) \right) \right]. \quad (2)$$

where $\rho_{i,t} = \frac{\pi_{\theta}(o_{i,t}|q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q, o_{i,<t})}$ denotes the probability ratio at step t , and β is the coefficient for the KL divergence regularization. Crucially, $\hat{A}_{i,t}$ represents the advantage computed via group normalization. Specifically, given the rewards $\mathbf{r} = \{r_1, \dots, r_G\}$ for the group of outputs, the advantage is derived as:

$$\hat{A}_{i,t} = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r}) + \epsilon}, \quad (3)$$

where the baseline is estimated solely from the group scores, obviating the need for a critic model.

4.2 Reward Function Design

We define the total reward $R(p)$ for a generated program p . This total reward comprises the sum of a format consistency reward and a performance-driven efficiency reward:

$$R(p) = r_{\text{format}}(p) + r_{\text{eff}}(p). \quad (4)$$

The first component is the format reward $r_{\text{format}}(p)$. This term acts as a sparse signal ensuring that the generated code adheres to the strict formatting requirements specified in the prompt. Specifically, we assign a reward score of 0.05 if the system successfully extracts a code block and identifies a defined calculator function within that block.

The core optimization objective is encapsulated in the efficiency reward $r_{\text{eff}}(p)$. We compute this metric exclusively for programs that pass the functional correctness tests; the reward is zero otherwise.

To emphasize substantial speedups, we introduce a margin parameter $\tau \geq 0$ and shift the efficiency score by subtracting τ , so that minor improvements below the margin receive no positive reward. In our training runs, we set $\tau = 0.5$. To stabilize training, we further constrain the reward within $[0, 2]$. The

Model	OEISPerf			HumanEval		MBPP		LiveCodeBench v5
	RAR	RNS	NVG	HE	HE+	MBPP	MBPP+	Pass@1
Qwen2.5-Coder-3B-Instruct	2.3	21.6	70.2	83.5	78.7	73.8	62.7	21.6
Qwen2.5-Coder-3B-Instruct-OEISPerf	12.1	43.0	44.5	85.4	79.3	75.1	64.3	23.3

Table 3: The performance of Qwen2.5-Coder-3B-Instruct fine-tuned on OEISPerf compared with the base Qwen2.5-Coder-3B-Instruct. RAR denotes the reference attainment rate, RNS the reference-normalized speedup score, and NVG the no valid gain rate on OEISPerf.

final efficiency reward is:

$$r_{\text{eff}}(p) = \text{clip}\left(\frac{\ln t(p_{\text{ineff}}) - \ln t(p)}{\ln t(p_{\text{ineff}}) - \ln t(p_{\text{ref}})} - \tau, 0, 2\right). \quad (5)$$

This formulation incentivizes the model to generate functionally correct programs while prioritizing large runtime improvements that approach or exceed the execution speed of the reference implementations.

4.3 Evaluation Benchmarks

We evaluate the GRPO-finetuned model, denoted as Qwen2.5-Coder-3B-Instruct-OEISPerf, on OEISPerf. To ensure fine-tuning preserves foundational coding abilities, we also evaluate on HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCodeBench (Jain et al., 2025). For HumanEval and MBPP, we adopt the EvalPlus (Liu et al., 2023) evaluation framework, which augments each task with additional test cases and yields the corresponding HumanEval+ and MBPP+ scores. Unless otherwise specified, we report pass@1 on all benchmarks.

4.4 Evaluation Results

Table 3 compares the base model Qwen2.5-Coder-3B-Instruct with its GRPO-finetuned counterpart, denoted as Qwen2.5-Coder-3B-Instruct-OEISPerf. On OEISPerf, fine-tuning substantially improves optimization behavior: the reference attainment rate rises from 2.3% to 12.1%, the reference-normalized speedup score increases from 21.6 to 43.0, and the no valid gain rate drops from 70.2% to 44.5%. At the same time, performance on out-of-distribution code benchmarks also improves slightly. In summary, these results suggest that reinforcement learning on OEISPerf can meaningfully strengthen the program optimization ability of the LLM while largely preserving, and in some cases modestly enhancing, its general code generation capability.

4.5 Hidden-State Alignment Analysis

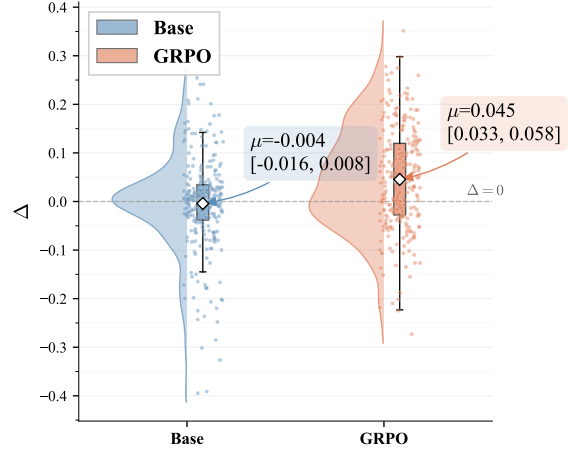


Figure 5: Distribution of representational shift Δ across the test split for the base model Qwen2.5-Coder-3B-Instruct and the GRPO-finetuned model Qwen2.5-Coder-3B-Instruct-OEISPerf. For each instance, Δ measures the within-model gain in cosine similarity from the initial inefficient program p_{ineff} to the generated program p_{gen} with respect to the efficient reference program p_{ref} . Half-violin plots show the empirical density, box plots summarize the interquartile range and median, and white diamonds mark the dataset mean. We also report 95% bootstrap confidence intervals for the mean shift; the interval for the Base model slightly overlaps zero ($[-0.016, 0.008]$), whereas the GRPO-finetuned model has a clearly positive interval ($[0.033, 0.058]$), indicating a consistent movement of representations toward the reference solutions.

To better understand what changes after finetuning, we analyze how the internal representations of programs differ between the base model and the GRPO-finetuned model. For each program, we take the final hidden state of the last token as its representation. Due to the autoregressive nature of the model, this state attends to the entire prefix and thus encompasses information from the whole input text.

For every instance, we encode three programs: the inefficient implementation, the efficient reference solution, and the code produced by the

Model	$\cos(p_{ineff}, p_{ref})$	$\cos(p_{gen}, p_{ref})$	Δ
Qwen2.5-Coder-3B-Instruct	0.771 ± 0.072	0.766 ± 0.113	-0.004
Qwen2.5-Coder-3B-Instruct-OEISPerf	0.746 ± 0.077	0.791 ± 0.102	0.045

Table 4: Cosine similarity to the efficient reference program p_{ref} computed in each model’s own representation space, reported as mean $\pm$ standard deviation on test split. The relative representational shift is defined as $\Delta = \cos(p_{gen}, p_{ref}) - \cos(p_{ineff}, p_{ref})$. We assess optimization behaviour through the within-model change Δ , which captures how generation moves the representation of a program toward the reference solution.

model. Let h_{ineff} , h_{ref} , and h_{gen} denote their last-token hidden states. We compute cosine similarities $\cos(h_{gen}, h_{ref})$ and $\cos(h_{ineff}, h_{ref})$ within the latent space of each model and define a differential alignment score:

$$\Delta = \cos(h_{gen}, h_{ref}) - \cos(h_{ineff}, h_{ref}). \quad (6)$$

This quantity measures how much the generated program increases alignment with the target representation relative to the initial code inside the same embedding space.

As summarized in Table 4, the Base model exhibits a slight negative shift in Δ , which means that its hidden representation for the generated program is slightly further from the target than the representation of the initial implementation. By contrast, the GRPO-finetuned model shows a substantially larger and consistently positive Δ , with Figure 5 indicating that most instances experience a clear movement of the generated representation toward the reference solution. Taken together, these results indicate that after GRPO finetuning, the model’s generated programs are represented more similarly to efficient reference solutions.

5 Related Work

LLMs for Code. LLMs have demonstrated strong capabilities across a wide range of natural language processing tasks, and this progress has rapidly transferred to code generation. Early large-scale efforts such as Codex demonstrated that LMs fine-tuned on publicly available code from GitHub can generate functionally correct programs from docstrings and power developer tools in practice (Chen et al., 2021). AlphaCode further showed that large-scale sampling and filtering can produce competitive-programming solutions (Li et al., 2022). Motivated by these advances, subsequent work has focused on developing code-specialized language models through large-scale pretraining and post-training. Representative open releases include StarCoder2 (Lozhkov et al., 2024),

DeepSeek-Coder and its Mixture-of-Experts successor DeepSeek-Coder-V2 (Guo et al., 2024; DeepSeek-AI et al., 2024), CodeGemma (Zhao et al., 2024) built upon Gemma with code-focused adaptation, and Qwen2.5-Coder (Hui et al., 2024), which reports strong results on code generation, code completion, and program repair. In parallel with these model advances, recent research has increasingly emphasized evaluation beyond functional correctness, including performance considerations, which aligns with the focus of OEISPerf.

Code Benchmarks. To evaluate functional correctness in code generation, researchers have proposed a range of foundational benchmarks. Representative examples include HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which measure the ability of language models to synthesize correct programs from natural language specifications. Subsequent efforts such as EvalPlus (Liu et al., 2023) expanded the test suites of these foundational datasets to rigorously ensure functional correctness and mitigate false positive evaluations. To address the growing concern of data contamination, LiveCodeBench (Jain et al., 2025) continuously curates problems from recent competitive programming contests to evaluate models on dynamically updated and unseen tasks.

Beyond functional correctness, recent work has increasingly emphasized the execution efficiency of the generated programs. EffiBench (Huang et al., 2024), EffiBench-X (QING et al., 2025), and Mercury (Du et al., 2024) introduced comprehensive benchmarks to measure the runtime and memory efficiency of code generated from natural language prompts across various programming languages.

Recent work such as PIE (Shypula et al., 2024) curates a code-to-code dataset of performance-improving edits made by human programmers across thousands of competitive programming submissions. However, many earlier paired datasets are mined from human submission trajectories and primarily reflect repair-style improve-

ments. The improvements are largely dominated by engineering-level micro-optimizations. Distinct from previous research, OEISPerf shifts the evaluation focus from statement-level edits toward algorithmic restructuring to assess the inductive reasoning capabilities of LLMs in program optimization tasks.

6 Conclusion

In this work, we introduced OEISPerf, a code-to-code benchmark for program optimization built from alternative implementations of OEIS integer sequences. Our analysis of tree edit distances and optimization potential shows that most instances require large structural changes and admit substantial speedups. Evaluating 15 LLMs reveals that LLMs often fall short of the efficient references. Finally, GRPO-based reinforcement learning on OEISPerf noticeably strengthens optimization behavior, while slightly improving performance on standard code-generation benchmarks.

Limitations

OEISPerf has two main limitations. First, our dataset construction uses DeepSeek-V3.2 to synthesize candidate implementations from extracted computation specifications, which may introduce model-specific construction bias despite subsequent validation and filtering. Second, although all programs are evaluated in a fixed runtime environment with repeated measurements to reduce variance, wall-clock measurements may still be affected by residual system-level variability, such as cache state, CPU scheduling, and frequency fluctuations. Since most OEISPerf instances exhibit substantial optimization gaps, such variability is unlikely to materially affect our main conclusions.

References

- Anthropic. 2025a. [Introducing Claude Haiku 4.5](#).
- Anthropic. 2025b. [Introducing Claude Sonnet 4.5](#).
- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *CoRR*, abs/2108.07732.
- ByteDance Seed. 2025. [Seed1.8](#).
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. [Evaluating large language models trained on code](#). *CoRR*, abs/2107.03374.
- DeepSeek-AI. 2025. [Deepseek-v3.2: Pushing the frontier of open large language models](#). *CoRR*, abs/2512.02556.
- DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, et al. 2024. [Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence](#). *CoRR*, abs/2406.11931.
- Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. [Mercury: A code efficiency benchmark for code large language models](#). In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.
- Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022. [Deeppperf: A deep learning-based approach for improving software performance](#). *CoRR*, abs/2206.13619.
- Gemini Team. 2025. [Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities](#). *CoRR*, abs/2507.06261.
- GLM. 2025. [GLM-4.5: agentic, reasoning, and coding \(ARC\) foundation models](#). *CoRR*, abs/2508.06471.
- Google DeepMind. 2025. [Gemini 3 flash model card](#).
- 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. 2024. [Deepseek-coder: When the large language model meets programming - the rise of code intelligence](#). *CoRR*, abs/2401.14196.
- Dong Huang, Yuhao Qing, Weiye Shang, Heming Cui, and Jie Zhang. 2024. [Effibench: Benchmarking the efficiency of automatically generated code](#). In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. [Qwen2.5-coder technical report](#). *CoRR*, abs/2409.12186.
- Nam Huynh and Beiyu Lin. 2025. [Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications](#). *CoRR*, abs/2503.01245.

- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). In *The Thirteenth International Conference on Learning Representations*.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. [A survey on large language models for code generation](#). *ACM Trans. Softw. Eng. Methodol.*, 35(2).
- Kimi Team. 2025. [Kimi K2: open agentic intelligence](#). *CoRR*, abs/2507.20534.
- Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. [Competition-level code generation with alphacode](#). *CoRR*, abs/2203.07814.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. [Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation](#). In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Llama Team. 2024. [The llama 3 herd of models](#). *CoRR*, abs/2407.21783.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. [Starcoder 2 and the stack v2: The next generation](#). *CoRR*, abs/2402.19173.
- OpenAI. 2025a. [gpt-oss-120b & gpt-oss-20b model card](#). *CoRR*, abs/2508.10925.
- OpenAI. 2025b. [Introducing GPT-5.2](#).
- Yuhao QING, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu Ming Yiu, Dong HUANG, See-Kiong Ng, and Anh Tuan Luu. 2025. [Effibench-x: A multi-language benchmark for measuring efficiency of LLM-generated code](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.
- Qwen Team. 2025. [Qwen3 technical report](#). *CoRR*, abs/2505.09388.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. [Deepseekmath: Pushing the limits of mathematical reasoning in open language models](#). *CoRR*, abs/2402.03300.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. [Hybridflow: A flexible and efficient RLHF framework](#). In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 1279–1297. ACM.
- Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. [Learning performance-improving code edits](#). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- N. J. A. Sloane. 2023. ["a handbook of integer sequences" fifty years later](#). *Preprint*, arXiv:2301.03149.
- Kaizhong Zhang and Dennis E. Shasha. 1989. [Simple fast algorithms for the editing distance between trees and related problems](#). *SIAM J. Comput.*, 18(6):1245–1262.
- Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, et al. 2024. [Codegemma: Open code models based on gemma](#). *CoRR*, abs/2406.11409.

A On-Line Encyclopedia of Integer Sequences

The OEIS is a curated and publicly accessible repository of integer sequences with rich mathematical metadata. Each sequence is assigned a unique A number that consists of the letter A followed by six digits. The OEIS home page reports 393586 sequences in early 2026. Each entry records initial terms and a brief definition, together with structured fields that may include comments, formulas, recurrences, references, links, and programs for term generation.

The OEIS database is distributed under the Creative Commons Attribution Share-Alike 4.0 license (CC-BY-SA-4.0). We comply with attribution and ShareAlike requirements, and use the data for academic research.

Figure 6 shows a screenshot of a typical OEIS sequence entry page.

B Version Information of the Models

As shown in Table 5, we detail the specific versions or access identifiers of all models considered in this paper. For open-source models, we additionally report the license specified by the corresponding model repository.

[login](#)

The OEIS is supported by [the many generous donors to the OEIS Foundation](#).

0 1 3 6 2 7
: 13
: 20
23 12
10 22 11 21

THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES®

founded in 1964 by N. J. A. Sloane

[Hints](#)

(Greetings from [The On-Line Encyclopedia of Integer Sequences!](#))

A000248
Expansion of e.g.f. exp(x*exp(x)).
108

(Formerly M2857 N1148)

1, 1, 3, 10, 41, 196, 1057, 6322, 41393, 293608, 2237921, 18210094, 157329097, 1436630092, 13810863809, 139305550066, 1469959371233, 16184586405328, 185504221191745, 2208841954063318, 27272621155678841, 348586218389733556, 460522387997411873

([list](#); [graph](#); [refs](#); [listen](#); [history](#); [text](#); [internal format](#))

OFFSET
0,3

COMMENTS

Number of forests with n nodes and height at most 1.
Equivalently, number of idempotent mappings f from a set of n elements into itself (i.e., satisfying f o f = f). - [Robert FERREOL](#), Oct 11 2007
In other words, a(n) = number of idempotents in the full semigroup of maps from [1..n] to itself. [Tainiter]
a(n) is the number of ways to select a set partition of {1,2,...,n} and then designate one element in each block (cell) of the partition.
Let set B have cardinality n. Then a(n) is the number of functions f:D->C over all partitions {D,C} of B. See the example in the Example Section below. We note that f:empty set->B is designated as the null function, whereas f:B->empty set is undefined unless B itself is empty. - [Dennis P. Walsh](#), Dec 05 2013
In physics, a(n) would be interpreted as the number of projection operators P on S_n, i.e., ones satisfying P^2 = P. Example: a particle with a half-integer spin s has a spin space with 2s+1 base states which admits a(2s+1) linear projection operators (including the identity). These are important because they satisfy the operator identity exp(zU) = 1+(exp(z)-1)*U, valid for any complex z. - [Stanislav Sykora](#), Nov 03 2016

REFERENCES

L. Comtet, *Advanced Combinatorics*, Reidel, 1974, p. 91.
N. J. A. Sloane, *A Handbook of Integer Sequences*, Academic Press, 1973 (includes this sequence).
N. J. A. Sloane and Simon Plouffe, *The Encyclopedia of Integer Sequences*, Academic Press, 1995 (includes this sequence).
R. P. Stanley, *Enumerative Combinatorics*, Cambridge, Vol. 2, 1999; see Problem 5.32(d).

LINKS

Alois P. Heinz, [Table of n, a\(n\) for n = 0..541](#) (first 101 terms from T. D. Noe)

Figure 6: An example OEIS entry page. The identifier in this page is “A000248”. This webpage includes the number sequence, number sequence offsets, number sequence references, number sequence links to other supplementary information, mathematical explanations, cross-references with other number sequences, and so on.

C AST distance implementation details.

For the structural analysis in Table 1, we parse each program into a Python AST and compute tree edit distance using the Zhang–Shasha algorithm. To better reflect structural transformation rather than superficial lexical variation, we construct the edit-distance trees using AST node types as labels, without encoding identifier names or literal values. This design makes the metric less sensitive to cosmetic edits such as variable renaming, and instead emphasizes changes in program organization, expression structure, and control-flow composition. As a result, the reported distance is intended to capture the degree of syntactic restructuring between p_{ineff} and p_{ref} .

D OEISPerf Statistics

Table 6 reports the descriptive statistics of the dataset partitions employed in this work, namely the training, validation, and test splits. For each partition, *Min*, *Max*, *Average*, and *Median* correspond to the minimum, maximum, mean, and median

number of terms per instance, respectively. The numbers reported for *train*, *valid*, and *test* represent the total number of instances in each split.

E Evaluation

E.1 Runtime Environment

All runtime measurements are obtained on a server with x86_64 architecture and dual socket Intel Xeon Platinum 8352V CPUs at 2.10 GHz. Each socket hosts 32 physical cores and 64 hardware threads, for 64 physical cores and 128 hardware threads in total. The cache hierarchy comprises 3 MiB L1 data cache and 2 MiB L1 instruction cache across 64 instances, 80 MiB L2 cache across 64 instances, and 108 MiB L3 cache across 2 instances. All reported wall clock times are measured in this environment.

E.2 Evaluation Implementation Details

Table 7 outlines the key hyper-parameters used in our evaluation framework.

Open-source Model	Provider	Link/Version	License
GLM 4.7	Zhipu AI	https://huggingface.co/zai-org/GLM-4.7	MIT
Kimi-K2	Moonshot AI	https://huggingface.co/moonshotai/Kimi-K2-Thinking	Modified MIT
Qwen3-14B	Qwen Team	https://huggingface.co/Qwen/Qwen3-14B	Apache-2.0
GPT-OSS-20B	OpenAI	https://huggingface.co/openai/gpt-oss-20b	Apache-2.0
DeepSeek-V3.2	DeepSeek-AI	https://huggingface.co/deepseek-ai/DeepSeek-V3.2	MIT
Llama-3.1-8B-Instruct	Meta	https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct	Llama 3.1 Community License
Llama-3.2-3B-Instruct	Meta	https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct	Llama 3.2 Community License
Qwen2.5-Coder-3B-Instruct	Qwen Team	https://huggingface.co/Qwen/Qwen2.5-Coder-3B-Instruct	Qwen Research License
Closed-source Model	Provider	Link/Version	
Gemini-3-Flash	Google	gemini-3-flash-preview	
Gemini-2.5-Flash	Google	gemini-2.5-flash	
Seed-1.8	ByteDance Seed	doubao-seed-1-8-251228	
GPT-5.2	OpenAI	gpt-5.2-2025-12-11	
Qwen3-Max	Qwen Team	qwen3-max-2025-09-23	
Claude-Sonnet-4.5	Anthropic	claude-sonnet-4.5	
Claude-Haiku-4.5	Anthropic	claude-haiku-4.5	

Table 5: Version and license information of the models evaluated on OEISPerf. The license column is reported for open-source models according to the corresponding model repositories.

Split	#Instances	Min	Max	Average	Median
Train	2762	4	120	36.24	28.0
Valid	240	5	105	36.50	25.5
Test	265	6	105	38.42	28.0

Table 6: Descriptive statistics of the dataset splits used in this work. #Instances denotes the number of instances in each split, while Min, Max, Average, and Median represent the minimum, maximum, mean, and median number of terms per instance, respectively.

Resource limits and sandboxing. All correctness verification and runtime evaluation runs execute in separate subprocesses. We cap the memory using `prlimit -as` (memory limit M GB), and enforce a wall-clock timeout of 3400s per subprocess.

Software Environment. All generated Python programs were executed and validated in a standardized environment running Python 3.13. Mathematical functions and generation requirements were handled using SymPy version 1.14.0.

E.3 Results

Figure 7 shows the complete results of LLMs.

E.4 Multi-sample Evaluation

Table 8 reports reference attainment pass@k results with $n = 10$ samples per instance for representative LLMs.

F Reinforcement Learning

F.1 Training Configuration

We fine-tune the model with GRPO for a total of six epochs, using LoRA-based parameter-efficient adaptation with rank 32 and scaling factor 64 applied to all linear layers in the transformer. The

policy is optimized with the AdamW optimizer with a learning rate of 3×10^{-5} , weight decay of 0.01, and Adam momentum coefficients $(\beta_1, \beta_2) = (0.9, 0.999)$, together with global gradient norm clipping at 1.0 to stabilize training. We employ a cosine learning-rate decay schedule with a warmup phase covering 5% of the total training steps, after which the learning rate is gradually annealed following the cosine schedule. All GRPO fine-tuning experiments were conducted on a server equipped with NVIDIA A10 GPUs. The total computational budget for fine-tuning the model was approximately 70 GPU hours.

F.2 Training Process

As shown in Fig. 8, we present the overall reward trajectory of the Qwen2.5-Coder-3B-Instruct model during training. The blue curve denotes the instantaneous reward at each training step, while the orange curve shows the moving average of rewards, which provides a smoother view of the overall trend.

F.3 Out-of-distribution Code Benchmark Licenses

Table 9 summarizes the licenses of the out-of-distribution code benchmarks and we use these

Symbol / Parameter	Value in our evaluation implementation
k (The last k items of the input)	$k = 7$
m (Times per benchmark repeat)	$m = 50$
r (Repeat rounds)	$r = 7$
M (Memory limit)	$M = 0.5$ GB
subprocess timeout	timeout = 3400 s

Table 7: Key evaluation hyper-parameters and implementation choices.

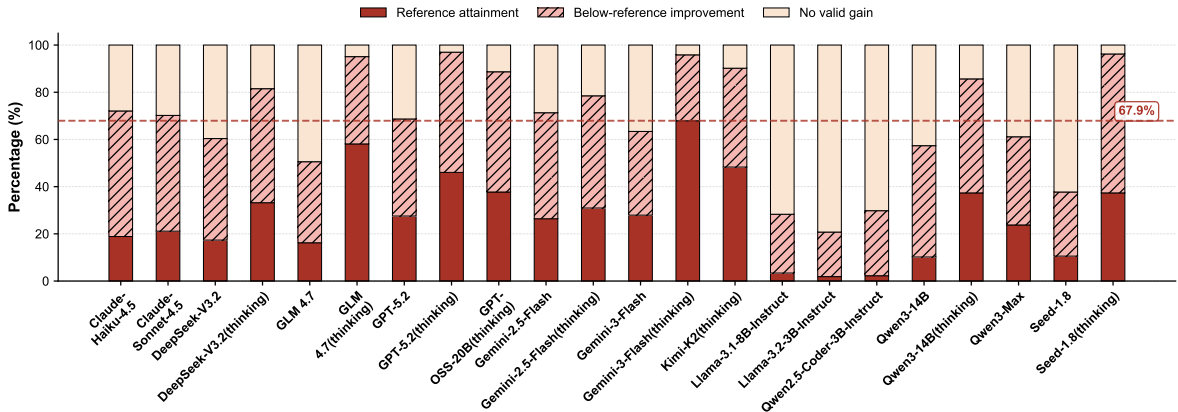


Figure 7: Performance distribution of all LLMs on OEISPerf.

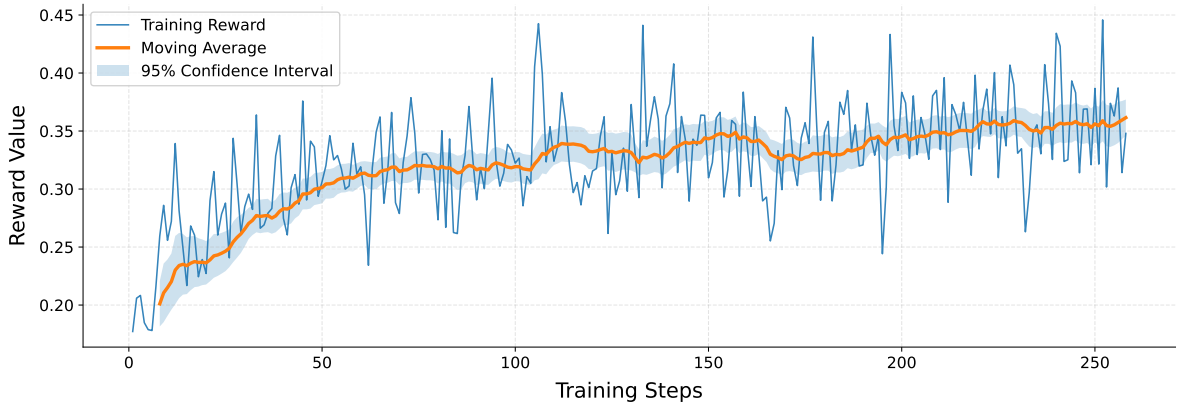


Figure 8: Overall reward values of the Qwen2.5-Coder-3B-Instruct model during training.

Model	Pass@1	Pass@3	Pass@5
Seed-1.8 (thinking)	38.7	55.2	61.6
Seed-1.8	10.5	17.7	22.0
GPT-OSS-20B (thinking)	34.3	53.9	63.5

Table 8: Reference attainment pass@k results with $n = 10$ samples per instance for representative LLMs.

Benchmark	License
HumanEval	MIT
HumanEval+	Apache-2.0
MBPP	CC BY 4.0
MBPP+	Apache-2.0
LiveCodeBench	MIT

Table 9: Licenses of the out-of-distribution benchmarks used in our evaluation.

benchmarks only for research evaluation and do not redistribute modified benchmark data.

G Prompt

Table 10, Table 11, and Table 12 summarize the prompts used in our pipeline: code generation from OEIS formulas, automatic program repair

when tests fail, and program optimization during evaluation.

Prompt of Code Generation

You are a Python master. Your task is to faithfully implement the given formula, rather than performing any mathematical derivation or simplification. Functions such as cos, sin, log, and exp can all be implemented using the corresponding functions in the 'sympy' library. In particular, the implementation of generating functions must use the 'sympy' library.

Note: ****Do not optimize the calculation process. Stay faithful to the formula, and do not consider the time complexity of computing too many terms!****

Write Python code to compute the sequence $a(n)$.

Uniform requirements (very important):

1. You must define and only define the 'calculator' function as the function entry point:

```
'''python
def calculator(n: int) -> int:
    ...
'''
```

2. The use of any caching to optimize calculations is strictly prohibited!

3. You are allowed to write a small number of helper functions, but please implement them ****inside the 'calculator' function****;

4. You must include one and only one function definition named 'calculator'. Do not write any test code, 'print' statements, or constructs like 'if __name__ == "__main__":'!

5. Please wrap the output code in a '''python code block!

Below is the formula description **from** the OEIS %F line. Please interpret it into a Python function to compute $a(n)$:

Original %F line (prefix removed):
{FORMULA}

Context:
Sequence ID: {ID}
Name (%N): {NAME}
Offset (%O): a={OFFSET_A}, b={OFFSET_B}
First four terms (starting **from** offset): {TERMS}

Please output the Python code that meets the requirements.

Table 10: Code-generation prompt used in dataset construction. The prompt also provides OEIS context (sequence ID, name, and %O offset a,b where a is the index of the first listed term and b is OEIS's internal second offset used for ordering, i.e., the position of the first term with $|a(n)| \geq 2$).

Prompt of Program Repair.

You are a Python master. Your task is to faithfully implement the given formula, rather than performing any mathematical derivation or simplification. Functions such as cos, sin, log, and exp can all be implemented using the corresponding functions in the 'sympy' library. In particular, the implementation of generating functions must use the 'sympy' library.

Note: ****Do not optimize the calculation process. Stay faithful to the formula, and do not consider the time complexity of computing too many terms!****

Write Python code to compute the sequence a(n).

Uniform requirements (very important):

1. You must define and only define the 'calculator' function as the function entry point:

```
'''python
def calculator(n: int) -> int:
    ...
'''
```

2. The use of any caching to optimize calculations is strictly prohibited!
3. You are allowed to write a small number of helper functions, but please implement them ****inside the 'calculator' function****;
4. You must include one and only one function definition named 'calculator'. Do not write any test code, 'print' statements, or constructs like 'if __name__ == "__main__":'!
5. Please wrap the output code in a **'''python code block!**

Previously, you generated code based on the instructions below, but an error occurred during automated testing.

Please fix the code based on your understanding of the error message and output it again.

```
===== Previous Instructions =====
{ORIGINAL_PROMPT}

===== Your Previously Generated Code =====
{INCORRECT_CODE}

===== Automated Testing Error Message =====
{ERROR_MESSAGE}

===== Context =====
Sequence ID: {ID}
Name (%N): {NAME}
Offset (%O): a={OFFSET_A}, b={OFFSET_B}
First four terms (starting from offset): {TERMS}
```

Please output the corrected, complete Python code (you must still only define a single 'def calculator(n)' function), and do not include any explanatory text or Markdown.

Table 11: Program-repair prompt for failed candidates: when previously generated code INCORRECT_CODE does not execute or fails validation, the model receives the original generation instructions, the faulty code, and the automated testing error message. It must output a corrected, complete implementation that preserves the same constraints, using the provided OEIS metadata (ID, name, offset a,b, and initial terms) to resolve errors, and returning only the fixed code for re-validation.

Prompt of Evaluation
Your task is to optimize the provided sequence calculation code, focusing on reducing time complexity to achieve extreme performance. Requirements: 1. Code must be enclosed in a <code>block</code>. 2. The function signature must strictly adhere to <code>def calculator(n: int) -> int:</code>. 3. The code block must strictly exclude testing code, print statements, or <code>if __name__ == "</code> <code>__main__":</code>. Code: <pre>'''python {CODE} '''</pre>

Table 12: The prompt used for evaluation.