
# LLM-VeriOpt: Verification-Guided Reinforcement Learning for LLM-Based Compiler Optimization

Xiangxin Fang
*University of Edinburgh*
*Queen Mary University of London*[*]
United Kingdom
jp2019213661@qmul.ac.uk

Jiaqin Kang
*Queen Mary University of London*
United Kingdom
k.jiaqin@se24.qmul.ac.uk

Rodrigo Rocha
*University of Edinburgh*[*]
United Kingdom

Sam Ainsworth
*University of Edinburgh*
United Kingdom
sam.ainsworth@ed.ac.uk

Lev Mukhanov
*Queen Mary University of London*
United Kingdom
l.mukhanov@qmul.ac.uk

***Abstract*—Large Language Models (LLMs) for compiler optimization have recently emerged as a frontier research direction, with many studies demonstrating their potential to automate and improve low-level code transformations. While various techniques have been proposed to enhance LLMs' ability to optimize LLVM IR or assembly code, ensuring the semantic equivalence of transformed instructions remains a fundamental prerequisite for safe and effective performance improvement. At the same time, code generated by LLMs is often so far away from being correct that it is very difficult to work out how to improve them to proceed in generating optimizations using the output.**

**In this work, we present LLM-VeriOpt, a novel reinforcement-learning methodology that incorporates feedback from a formal verifier, Alive2, to guide the training of a small-scale model, Qwen-3B. This facilitates the use of Guided Reinforcement via Group Relative Policy Optimization (GRPO), using semantic-equivalence signals from the Alive2 formal verification tool as part of the reward function. This allows the model to self-correct based on observing and subsequently learning to give correctness feedback during training, giving high code coverage by successfully transforming large amounts of code, while also optimizing it significantly.**

**We demonstrate our technique by designing an LLM-based peephole optimizer over LLVM-IR. Our method significantly improves the correctness of IR optimizations versus the base LLM Qwen-3B applied with just a prompt and no fine-tuning — achieving a 5.4× improvement in code successfully modified. The resulting model produces verifiably correct output 90% of the time, comfortably outperforming larger state-of-the-art LLMs, including Meta's LLM Compiler. This yields speedups of 2.3× over O0-optimized code, comparable to the handwritten LLVM `-instcombine` pass, and producing emergent optimizations that outperform it in 20% of cases.**

***Index Terms*—Optimizing Compilers, Large Language Models**

## I. Introduction

Modern compilers such as LLVM have accumulated decades of engineering expertise in order to deliver reliable optimizations over compiler intermediate representations (IR). However, they remain constrained by manually designed heuristics and fixed pass pipelines (e.g., LLVM -O3), which limit exploration of the vast transformation space and prevent convergence to globally optimal code [1].

To overcome this bottleneck, researchers have begun to explore machine-learning–based methods to automatically discover better optimization sequences, giving rise to the field of AI for Compiler Optimization. For example, CompilerGym [2] provides a scalable reinforcement learning environment for addressing the problem of LLVM pass ordering.

In recent years, Large Language Models (LLMs) have shown potential in generating novel compiler optimizations. Cummins et al. [3] conducted large-scale autotuning experiments to approximate the optimal pass sequence for each program, and then applied supervised fine-tuning (SFT) to train LLMs to predict these sequences. Their LLM-Compiler FTD model [4] outperformed LLVM -Oz in terms of binary size, with the 13-billion-parameter (13B) model producing smaller binaries in 61% of programs. Recent studies also show that incorporating reasoning [5], [6] into LLM prompts improves optimization accuracy and effectiveness.

Beyond pass ordering, finer-grained optimization tasks have been investigated. Several authors [7]–[9] attempt to go directly from language (e.g., C/C++) to target output assembly (e.g., x86). Wei et al. [7] leverage proximal policy optimization [10] reinforcement learning with Qwen-7B [11] to optimize assembly code. This achieves correctness in 96% of the cases (matches behavior of finite input-output samples) and achieves an average speedup of 1.47×, surpassing the gcc -O3 baseline, though relies on the actual -O3 code being included in the prompt, otherwise degrading to 0% accuracy. LLM Compiler's [12] Compiler Emulation task compiles (reports no syntax error) LLVM-IR to optimized LLVM-IR with 95.6% success, but with only 20% exact-match accuracy.

Such techniques suffer from a critical limitation: they do not strictly guarantee the semantic correctness of the transformed code. In most prior work [7], [12], correctness was assessed mainly through finite test suites (equivalence of input-output pairs over a set of samples, or I/O equivalence), which only approximate equivalence by testing a finite portion of the

[*]Affiliation at time of research.

979-8-3315-9288-2/26 © 2026 IEEE

input domain rather than providing a formal guarantee. This overestimates correctness; LLM-Vectorizer [13] demonstrates many cases where formal verification shows inequivalence of I/O-verified samples. In contrast, traditional compilers ensure deterministic equivalence, whereas LLMs are prone to hallucinations [14] that may generate invalid or inconsistent IR and assembly. This correctness gap represents the main barrier to deploying LLMs in practical compiler optimization.

LLM-Vectorizer [13] targets the task of vectorization. It points out that, because it transforms LLVM-IR (intermediate representation) to LLVM-IR, it is able to use the formal equivalency checker Alive2 [15] to reject incorrect samples, verifying 38% of the LLM's attempts as correct – and allows the use of existing compiler infrastructure to perform the transformations from input language to LLVM-IR, and LLVM-IR to assembly. This is a facilitating observation, as it means that, on tasks that output LLVM-IR, it is always possible to either verify correctness formally, or return the original code on a timeout or equivalence mismatch, meaning the LLM need no longer be trusted. Still, if the LLM is rarely able to output (verifiably) correct code, then the ability to optimize is severely impacted by the remaining unchanged code.

Reinforcement learning with verifiable rewards (RLVR) [16]–[18] has recently emerged as a promising method for enabling the self-improvement of LLMs in domains that demand structured reasoning. Within this framework, Group Relative Policy Optimization (GRPO) [19] has shown particular promise. Recent studies have demonstrated its effectiveness in tasks such as mathematical reasoning [19], where reaching correct solutions requires multi-step logical reasoning. Compiler optimization represents an analogous domain: transformations must follow rigorous reasoning principles and can be formally validated. Despite this strong alignment, GRPO has not yet been explored in this setting.

In this work, we propose LLM-VERIOPT, a reinforcement-learning framework that integrates Alive2's semantic equivalence checks into GRPO itself. Alive2 provides trustworthy verification feedback, which we leverage both as a reward signal and as learned diagnostic feedback incorporated into training prompts. This design enables our small-scale model (Qwen-3B [20], 3-billion parameters) to constantly learn from counterexamples throughout training, and, combined with also using Alive2 to verify LLVM-IR code at the backend [13], gives the potential to not only produce verified code, but to correct its own mistakes.

We focus on `-instcombine` [21] in LLVM: the compiler's peephole optimization pass. It performs algebraic simplifications and local transformations within each basic block, combining multiple instructions into more efficient forms. This pass is particularly well-suited for LLM-based approaches: it operates on short instruction sequences, mitigating the limitations of LLMs in handling large context windows [3]. Moreover, since *-instcombine* serves as the foundation for more advanced optimizations, ensuring correctness at this level is a prerequisite for scaling LLMs to broader compiler tasks. To evaluate and train our technique over code known to be challenging to compile correctly, we analyze C/C++ programs from both the LLVM and GCC test suites.

**The main goal** of this work is to bridge the correctness gap in LLM-based compiler optimization by developing a reinforcement learning framework that integrates formal verification feedback. Specifically, we show that, with assistance from formal verifiers, a small-scale LLM can reliably perform LLVM IR optimizations that are both semantically correct and efficient, thereby improving reliability beyond supervised fine-tuning and surpassing the performance of state-of-the-art models that are orders of magnitude larger.

**Our contributions** are as follows:

- We introduce LLM-VERIOPT, a novel framework for verified IR peephole optimization. It builds on Group Relative Policy Optimization (GRPO) and integrates Alive2 formal verification feedback directly into the reward function, thereby guiding reinforcement learning with semantic correctness guarantees.
- We construct a novel training scheme consisting of four progressive models: a MODEL-ZERO to analyze LLM-specific mistakes, a WARM-UP MODEL to bootstrap the process of training the LLM to generate code and diagnose its own mistakes, a MODEL-CORRECTNESS that uses reinforcement learning to optimize towards generating easily verifiable transformations via generating progressively higher quality diagnostics, and a MODEL-LATENCY that preserves highly verifiable transformations while generating emergent optimization capability.
- We evaluate on a dataset constructed of C/C++ programs from the LLVM and GCC test suites [22], [23]. We demonstrate that our method substantially improves the correctness and optimization performance of a small-scale model (Qwen-3B [20]), going from producing new correct code less than 20% of the time with only 0.2% speedup to one that produces new correct code 90% of the time and achieves 2.30× speedup. This comfortably outperforms larger state-of-the-art LLMs, and is comparable to the real handwritten `-instcombine` pass (2.39×) while demonstrating emergent capabilities by producing superior optimizations 20% of the time.

## II. PRELIMINARIES

### A. Compiler and LLVM IR:

LLVM [24], a widely used industrial compiler infrastructure, represents and transforms programs in an intermediate representation (LLVM IR) that is low-level enough to capture hardware details while remaining target-independent.

Peephole optimization is a classical compiler technique that replaces instruction sequences with logically equivalent but more efficient forms [25]. Typical categories include null sequence elimination, combining operations, algebraic simplifications, address mode optimizations, and the use of special-case instructions [26]. These local transformations are widely adopted across compilers, as they provide lightweight performance improvements while preserving correctness.

In LLVM, the canonical realization of peephole optimization is the `-instcombine` pass, which implements thousands of such algebraic and local transformations [24]. It is a fundamental component of LLVM's optimization pipeline and serves as the foundation for more advanced passes. Given the context-length limitations of current large language models [3], `-instcombine` provides an ideal setting for training LLMs: the optimization opportunities are local, and instruction sequences are relatively short, though complexity lies in the sheer number of optimizations included [27].

### *B. Supervised Fine-Tuning (SFT).*

SFT [28] refers to standard supervised training of the base model on (input, target) pairs, without reinforcement learning or preference optimization. In our setting, the target is the reference IR produced by LLVM's `-instcombine` pass. We use SFT as a baseline to evaluate how much improvement GRPO with Alive2 feedback can provide.

### *C. PPO vs. GRPO.*

Proximal Policy Optimization (PPO) [10] is the standard reinforcement-learning algorithm used in aligning large language models [29]. However, PPO relies on a separate value network to estimate absolute reward signals, which introduces additional model complexity and training overhead. In contrast, Group Relative Policy Optimization (GRPO) [19] eliminates the need for a value model by directly comparing multiple candidate outputs from the same prompt and assigning rewards based on their relative quality. This design makes GRPO particularly suited for tasks where feedback is binary or comparative, such as semantic-equivalence verification.

### *D. Semantic Equivalence Verification:*

Alive2 [15] is an automatic translation-validation tool for LLVM IR that provides formal guarantees of semantic equivalence between optimized and unoptimized programs. Alive2 relies on SMT (Satisfiability Modulo Theories) solvers which translate program instructions into algebraical expressions to check sematic equivalence using SAT or simplex-like algorithms [13], [30]–[32]. Alive2 is widely regarded [13], [30]–[32] in the research community as the most practical and robust translation validation tool for LLVM IR.

Beyond equivalence judgments, Alive2 also provides detailed diagnostic feedback, including syntax-level parsing errors and semantic mismatches between input and optimized IR. In our framework, we incorporate error messages into subsequent training prompts, enabling the model to learn directly from verification failures.

### *E. Prompting Setup (Generic Template).*

An illustration of the generic prompt template we use for all comparisons is shown in Fig. 1. We extend this where relevant for GRPO models that can generate their own intermediate Chain of Thought (CoT) steps as shown in section III-B. For supervised fine-tuning (SFT), which cannot reliably do this, and our GRPO baseline *without* prompt augmentation, the simple prompt here is used directly, unless otherwise noted.

**Generic Prompt**

```
<|im_start|>system
You are a helpful AI Assistant that optimizes LLVM IR.
This is a single-turn interaction.
IMPORTANT: The primary objective is to MINIMIZE
LATENCY while keeping the code complete, compilable,
semantically equivalent to the input, and preserving all
metadata.
Respond in the following format:
<answer>
- complete optimized LLVM IR
</answer>
<|im_end|>
<|im_start|>user
{Non-optimized IR}
<|im_end|>
<|im_start|>assistant
```

Fig. 1: Generic prompt template used for SFT and baseline GRPO (without augmentation).

We ask the LLM to optimize a full function at a time; likewise, we verify a full function at a time using Alive2. We found that liveness information between basic blocks is unavailable to the LLM if it only sees one block at a time. The same issue does not exist between functions, where input and output are controlled via strict interfaces, and so by processing larger blocks of code at once, we give the LLM sufficient information to solve the problem.

## III. LLM-VeriOpt: GRPO with Alive2 Verification

Here we present LLM-VeriOpt, our novel reinforcement-learning methodology. LLM-VeriOpt integrates Alive2's semantic equivalence checker into the GRPO framework, which enables the model (i) to optimize toward transformations that Alive2 can prove correct, and (ii) to directly learn from diagnostic-augmented samples that expose invalid optimizations. Building on this, we construct a hierarchy of models that are iteratively trained toward a latency-optimizing solution:

- A Model Zero to bootstrap the process of training augmented prompts with failure modes tailored towards the exact failures the underlying LLM makes. The intermediate output of the GRPO process, used to optimize towards accuracy, produces a wide dataset of failures, and combined with Alive2, the reasons for them.
- A Warm-Up Model generated via supervised fine-tuning (SFT) that mimics the process of providing Alive2 feedback on its own optimized output. This takes both the original `-instcombine` training set and Model Zero's failures, and tries to produce either a) the correct answer directly, or b) the wrong answer, an Alive2-style explanation, followed by the correct answer.
- A GRPO-optimized Model-Correctness that maximizes the chance of generating a correct transformation. This iteratively trains both the Alive2 feedback emulation

and the code-generation phases, such that error detection and correction gradually improve. This continues to the degree that ultimately no explicit error correction is needed, as it produces the right answer the first time by learning to avoid its common errors in the first place.
- Finally, a MODEL-LATENCY that preserves the accuracy of MODEL-CORRECTNESS while changing optimization criteria to explore and improve the performance of optimized code, while preserving semantic equivalence but removing incentives to exactly match `-instcombine`.

### A. Optimization towards Verifiably Correct Transformations

For each non-optimized IR, the policy generates multiple candidate transformations. We form pairs $(\text{Non-optimized}, \text{Candidate})$ and query Alive2 for semantic equivalence. Alive2 outcomes are incorporated into GRPO training as reward signals.

The overall reward is defined in a hierarchical manner.

$$r_i \;=\; t_i\Big(1 \;+\; a_i\big(1+m_i\big)\Big) \;+\; b_i, \tag{1}$$

where

$$\begin{aligned} t_i &\in \{0,1\}, && \text{completion-format compliance},\\ a_i &\in \{0,1\}, && \text{Alive2 semantic equivalence},\\ m_i &\in \{0,1\}, && \text{exact match with the reference IR},\\ b_i &\in [0,1], && \text{BLEU [33] similarity score.} \end{aligned}$$

The intuition is that a) correctness is highest priority to generate LLM-optimized code that can be externally verified with non-negligible chance, b) the function is continuous as a result of using BLEU scores [33], so that there is a steady incentive gradient for learning on, c) both similarity to InstCombine, and latency-reduction on code are used as tiebreakers whenever the code is correct, to disincentivize the input being returned as output. To be specific, outputs are required to satisfy the specified prompt format ($t_i$), semantic correctness is determined through Alive2 verification ($a_i$), and exact matches with the reference IR ($m_i$) are credited only when equivalence holds. The BLEU score [33] ($b_i$), based on the similarity of code and where a score of 1 indicates an exact match, offers feedback for partially correct outputs and thus alleviates the sparsity of purely discrete rewards, providing a continuous shaping signal to mitigate gradient starvation [34].

Producing correct (or optimized) code is not the same thing as producing identical code to `-instcombine`, and thus attempting to match it only forms part of the reward (via BLEU similarity and exact-match score). We reward alternatives that are (verifiably, via Alive2) correct as being superior to incorrect answers, and exact mimicry as superior to alternative semantic matches. Entirely incorrect output formats are least rewarded, and matches that are not semantically equivalent but are partly similar to `-instcombine` get partial credit by BLEU scoring.

This basic correctness reward is reused in various forms throughout our training pipeline: to incentivize producing correct code, to incentivize producing correct error messages when the code is wrong, and to incentivize producing correct *corrections* to wrong code in response to error messages. These are explored further in future sections. In addition, we also introduce rewards to *optimize* correct code to choose more preferable optimizations from the wide set of correct answers once we have a reliable LLM in section III-C3.

### B. Learning from Diagnostic Information

Beyond serving as a correctness oracle, Alive2 also produces diagnostic information that we incorporate into training. These messages allow us to emulate Alive2-style diagnostic feedback, which in turn guides the model toward generating more correct output. For each failed pair $(\textit{Non-optimized}, \textit{Candidate})$, Alive2 returns diagnostic feedback, which we combine with the input IR, the incorrect candidate, and the reference label (from `-instcombine`) to form diagnostic-augmented samples for training.

Fig. 2 shows this augmented prompt format. We extend the generic prompt from section II-E by introducing the `<think>` tag and embedding the Alive2-diagnostic-augmented samples (first attempt and error message if wrong) within this section, so that the model is exposed to diagnostic feedback during training. The output of this prompt can take two forms: either a direct, correct answer in a single attempt with an emulated Alive2 successful equivalence, or an incorrect answer, followed by a diagnostic error message. Both are followed by a final corrected answer. We prefer the correct answer first-time, but partially reward the wrong answer followed by correction, in order to bootstrap our GRPO models to progressively become more able to generate the former as they learn to diagnose bugs via generating the latter. This modified prompt is used for our two intermediate models: the WARM-UP MODEL and MODEL CORRECTNESS (section III-C2).

### C. Training Pipeline

Figure 3 presents a detailed overview of our training pipeline. The entire process starts from the pretrained `Qwen2.5-3B-Instruct` [20] base model. The pipeline is organized into three main stages and four models:

*1) Diagnostic-augmented Sample Generation (Stage 1):* A direct application of GRPO on the small-scale foundation model often collapses due to the sparsity of positive rewards: most candidate optimizations fail Alive2 checks and provide little useful gradient (section V-D). To address this, we deliberately run an initial GRPO phase using the generic prompt (fig. 1) to obtain a *preliminary policy*, denoted MODEL ZERO, following DeepSeek [35]. Although not a satisfactory optimizer, MODEL ZERO plays a crucial role as a *diagnostic-augmented sample generator*. Rather than using MODEL ZERO itself, we observe the GRPO training space while producing it and, through Alive2 validation, produce concrete diagnostic information based on the intermediate outputs. Importantly, these *diagnostic-augmented samples* are *model-adaptive*: they directly reflect the systematic weaknesses of Qwen-3B when applied to IR optimization. By reinjecting them into the training process as *Augmented Prompts* (fig. 2), with wrong code, diagnostics, and the correct output *(correction-augmented samples)* along

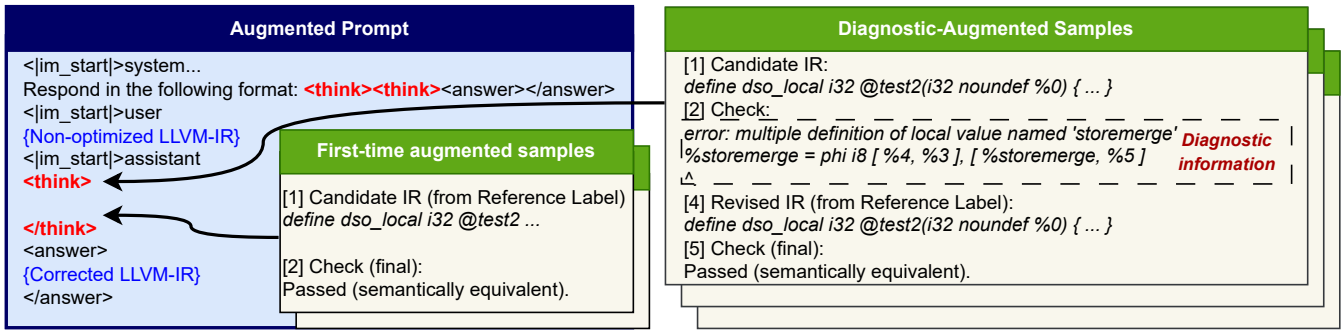



Fig. 2: We train the LLM to generate diagnostic error messages based on its initial mistakes, and produce corrected output based on the result. The initial dataset of *diagnostic-augmented samples* is generated by the wrong answers produced by an attempt to apply GRPO directly to Qwen-3B with the generic prompt (fig. 1), Alive2's response to them, and the actual intended `-instcombine` output. The wrong attempt, and the error diagnosis are inside the `<think>` block. The *corrected* answer is inside the `<answer>` block. We also include samples of the original O0-instcombine pairs, with the `-instcombine` answer inside the think block and as the output, to represent the preferred case when the LLM gets the answer right *first time*.

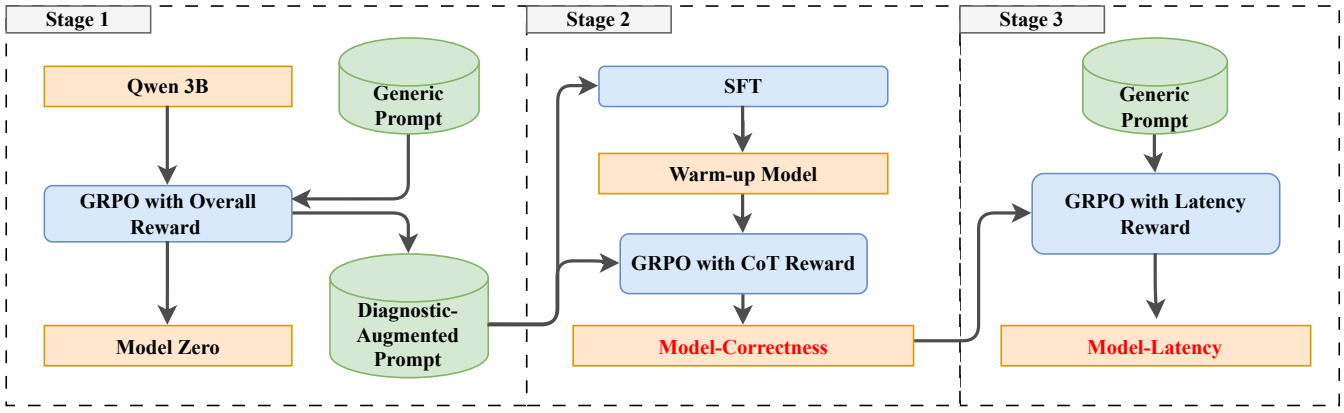



Fig. 3: The training pipeline: from MODEL ZERO (diagnostic-augmented sample generator) and WARM-UP MODEL (SFT on diagnostic-augmented samples), through GRPO with augmented prompts to obtain MODEL-CORRECTNESS, and subsequent incremental learning yielding the final MODEL-LATENCY.

with the original training data of O0-instcombine pairs *(first-time augmented samples)*, we enrich the otherwise sparse supervision with error signals that are maximally relevant to the model itself, laying the foundation for subsequent correctness-oriented training.

*2) Correctness-Oriented Training (Stage 2):* Even with augmented prompts, directly applying GRPO remains unstable because the foundation model lacks basic error-recognition capability. In particular, it cannot reliably distinguish invalid or semantically incorrect IRs, nor can it produce diagnostic information that explains why the output fails verification. We therefore introduce a *Warm-up stage*[1]: supervised fine-tuning (SFT) on the augmented prompts, in order to emulate the response of Alive2 inside the decision-making process. The WARM-UP MODEL equips the policy with rudimentary diagnostic skills and serves as a stronger initializer. The input to the Warm-up stage's SFT are the augmented samples described in fig. 2: both *first-time* augmented samples, which get the answer right immediately, and *correction-augmented* samples that get the answer wrong, diagnose it, and then correct it. For each code input, there will be one first-time augmented sample and potentially several correction-augmented samples depending on how many different ways the GRPO training of MODEL ZERO failed. The WARM-UP MODEL is not trained with any preference between these, outputting whichever of these several possible outputs is easiest for it to generate.

Building on this, we apply GRPO with augmented prompts, guiding the model to autonomously generate improved candidate IRs and improved validation of their semantics[2]. This stage culminates in MODEL-CORRECTNESS, a policy optimized to maximize Alive2-verified semantic equivalence. As shown in Figure 4(a), correctness steadily improves as GRPO progresses from the WARM-UP MODEL.

Just as in the training data, the correct answer can be derived either from getting the answer right immediately inside the `<think>` block (as in the first-time samples), or getting the answer wrong inside the think block, correctly diagnosing the issue, then fixing it (as in the correction samples). This creates a joint optimization between rewarding a) correct optimization in one shot, b) correct diagnostic information when the first

[1]This mirrors the strategy in AlphaGo [36], where supervised initialization provides a reliable policy prior to reinforcement learning.

[2]As new types of error appear as the LLM gets gradually better at producing code, Alive2 is constantly reconsulted and learned from.

attempt is wrong, and c) correct output on the second attempt. The principles follow the same reward function in section III-A. Inside the `<think>` tag, where an augmented prompt makes its first code attempt and a diagnosis if wrong, we assign a Chain of Thought (CoT) reward. Let $\mathcal{A}(S, T)$ denote the Alive2 judgment of the model's candidate IR $T$ against the source $S$, returning either **OK** (semantically equivalent) or **ERR** (not equivalent). The model also produces a self-diagnosis, including optional feedback text $F_{\text{model}}$, while Alive2 provides its diagnostic message $F_{\text{alive}}$.

The Chain of Thought(CoT) reward is defined as:

$$R = \begin{cases} 1, & \text{if both agree on } \textbf{OK}, \\ 0.5 + 0.5 \cdot \text{BLEU}(F_{\text{model}}, F_{\text{alive}}), & \text{if both agree on } \textbf{ERR}, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Thus, agreement on correctness yields full reward, agreement on errors yields a partial reward proportional to the similarity of explanations, and disagreement yields zero reward.

For the `<answer>` block, the score matches Equation (1) (section III-A). The final score sums these two components[3].

*3) Incremental Learning for Latency Optimization (Stage 3):* While MODEL-CORRECTNESS ensures semantic equivalence, it does not address runtime performance. To push further, we incrementally fine-tune MODEL-CORRECTNESS with a latency-oriented reward to generate MODEL-LATENCY. As illustrated in Figure 4(b), this stage drives the policy toward generating lower-latency code while maintaining correctness.

At this point, we stop using labeled `-instcombine` data in order to allow GRPO to explore policies that are potentially more optimal than the original `-instcombine`. We still maintain correctness in terms of a reward function that incentivizes only semantic equivalence to Alive2 rather than the similarity to `-instcombine` encoded in section III-A. The scale of reward is based on the relative speedup between baseline latency $t(P)$ (at `-O0`) and candidate latency $t(P')$:

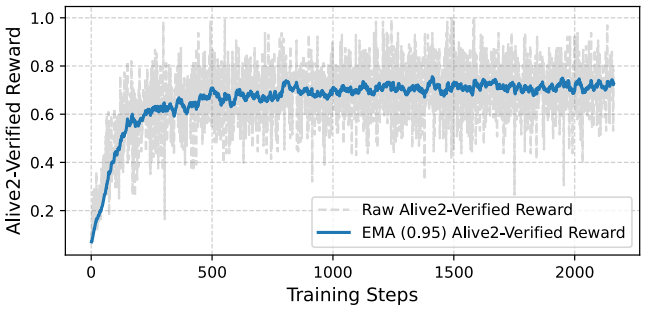
$$u = \frac{t(P)}{t(P')}, \qquad u \in (0, \infty). \tag{3}$$

$$r_{\text{lat}} = \begin{cases} 0, & \text{if } S = 0 \text{ or } u \leq 1, \\ \left(\min\left(1,\ \frac{u-1}{U_{\max}-1}\right)\right)^{\gamma}, & \text{if } S = 1 \text{ and } u > 1. \end{cases} \tag{4}$$

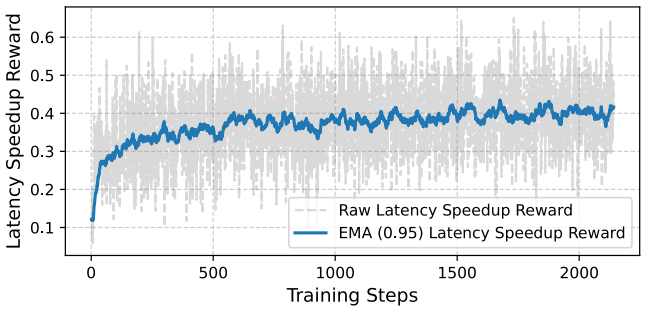
Here, $u$ is the speedup ratio, $S$ is the semantic equivalence check, $\gamma > 1$ a convex shaping factor, and $U_{\max}$ the saturation threshold. For reward normalization, we set $\gamma > 1$ to emphasize larger speedups, and set $U_{\max}$ as the 80th percentile of LLVM `-instcombine`'s speedups on the training set.

Output of Alive2-error emulation is also dropped as part of this model (but Alive2 is not dropped from the reward function), since we never directly use the error output and to avoid wasting generation capacity of the finite-parameter LLM model on useless output, returning to the original generic prompt (fig. 1) rather than the augmented error prompt (fig. 2) as output. However, we can infer from the lack of an accuracy drop (section V-D) that the error emulation that powers the GRPO of Model-Correctness, and lets it outperform MODEL ZERO, is preserved internally inside the model's capabilities.

[3]A wrong code sequence inside the `<think>` block followed by the correct error message, followed by the correct answer (matching the correctness-augmented samples) in theory gets the same score as getting the answer right the first time and repeating it in the answer block (matching the first-time augmented samples). However, since the former involves getting a tricky sequence of error messages correct, we find that the latter format, where the `<think>` code is correct, ends up being incentivized as the model becomes more able to correct its own errors, so the model becomes better at getting code right the first time. This allows us to then drop the `<think>` stage for the next model, MODEL-LATENCY.




(a) Correctness-oriented stage.




(b) Latency-oriented stage.

Fig. 4: Training dynamics of GRPO under different reward settings. The dashed line shows the raw latency speedup reward; the solid line shows the EMA-smoothed (0.95) curve.

## IV. EXPERIMENTAL SETUP

### A. Training and Test Set Generation

We train and evaluate LLM-VeriOpt by constructing a dataset from the LLVM [22] and GCC [23] test suites. These suites have long served as standard validation workloads for compiler research, as they are explicitly designed to cover a wide range of optimization patterns and to reveal hidden bugs in corner cases. We use these benchmarks to enable the model to cover diverse optimization combinations, including edge cases, and thereby improve its generalization ability, as well as to evaluate it over challenging code.

We first compile the source programs into LLVM IR using the `clang`/`clang++` frontend. The non-optimized IR is obtained by invoking `opt` with the `-O0` flag, while the optimized IR is generated with `opt -instcombine`, which applies a series of peephole optimizations. We then use `llvm-extract` to split IR into individual functions, so that each serves as a self-contained training example.

For training, we focus on the subset of functions whose `-O0` and `-instcombine` forms are proven semantically equivalent by Alive2. Pairs that are inequivalent[4], trigger undefined behavior, or cause verification timeouts are excluded[5]. For validation, we construct an independent dataset of 4,386 LLVM IR functions derived from the same GCC and LLVM test suites. This dataset is strictly isolated from the training set to avoid any data leakage, and its construction otherwise follows the methodology above.

### B. LLM Inference Strategy

We adopt greedy decoding [45] as our sole inference strategy, to to ensure that repeated inferences on the same input IR yield identical outputs, eliminating reproducibility issues (in e.g., stochastic or temperature-controlled strategies [46]). Our choice follows the setup in LLM-Compiler [4]. We utilize four common modifications to stabilize/simplify GRPO training:

1) **Removal of the KL penalty.** Recent studies (e.g., Open-Reasoner-Zero [47]) suggest that Kullback–Leibler (KL) divergence [48] penalty is not essential for training with GRPO. By eliminating the KL term, the policy is allowed to explore IR transformations more aggressively. Instead of KL regularization [49], we rely on gradient clipping to maintain stability during training.
2) **Single-update objective.** Since our training data is abundant and easy to collect, there is no need to perform multiple gradient updates on the same batch of rollouts. To avoid amplifying noise, we do not adopt the multi-update clipped surrogate objective [10]; hence, our objective reduces to the single-update formulation.
3) **Token-Level Loss Normalization.** In the original GRPO paper [19], the loss is first averaged within each sample and then averaged across samples, assigning equal weight to each sample regardless of its length. This introduces a length bias: long sequences are under-penalized and short responses under-rewarded, leading the model to prefer excessively long outputs. DAPO [50] highlights this and proposes a token-level normalization scheme, in which the loss is normalized by the total number of tokens across the global batch rather than by sequence length. This adjustment ensures that every token contributes equally, mitigating length bias.

### C. Metrics

We characterize IR optimizations along two dimensions: *correctness* and *efficiency*. These metrics serve dual roles: some are used purely for evaluation, while others also act as reward signals during training.

[4]As of writing, due to `-instcombine`'s complexity it is quite common for it to produce code that Alive2 can prove is an invalid transformation, with many open issues [37]–[40] on GitHub.

[5]Since prior studies have shown that large language models tend to degrade in performance when operating over very long context windows [41], we restrict the context window in 2048 tokens in our experiments. Specifically, we tokenize all IR using the Qwen-3b tokenizer and filter out samples with more than 2048 tokens. Similar practices have also been adopted in prior work on machine learning-based compiler optimization and neural lifting [3], [12], [42]–[44]. Finally, our final training set comprises 34,190 function pairs.

**Correctness.** Semantic equivalence is the strongest indicator of correctness in LLVM IR: when two IR programs are semantically equivalent, compilation is guaranteed to succeed. We rely on the `Alive2` validator to formally check equivalence between the input and optimized IR. Alive2 outcomes are categorized into four cases:

1) *Syntactic error*: invalid IR, non-compilable.
2) *Semantic error*: compilable but changes behavior.
3) *Inconclusive*: Alive2 cannot prove equivalence.
4) *Semantic equivalence*: formally proven equivalent.

**Efficiency.** Beyond correctness, we report three efficiency metrics to quantify the effectiveness of IR optimizations:

- **Estimated Latency:** Execution latency is estimated for each IR module on an AArch64 target. For each instruction, we query LLVM's `getInstructionCost(...,TCK_Latency)` API (LLVM 21.0.0git) to obtain estimated latency, and then sum all instructions to yield module-level latency[6].

- **Instruction Count (ICount):** Number of LLVM IR instructions in a module, reflecting program size at IR level.

- **Binary Size:** Following Cummins et al.'s LLM-Compiler methodology [4], we measure binary size as the on-disk size (in bytes) of the compiled object file. We sum the `.TEXT` and `.DATA` sections reported by `llvm-size`, while excluding the `.bss` section since it does not contribute to file size.

## V. Evaluation

We address the following research questions:

**RQ1 (Foundation Capability):** Can pre-trained foundation models effectively perform peephole optimization while preserving semantic equivalence, with no further fine-tuning? **A1:** While accuracy looks superficially high at 73.2%, we find that the vast majority of these cases result in the input being returned as output, producing *different correct* output from `-O0` in only 16.4% of cases.

**RQ2 (Optimization Effectiveness):** Do the optimizations produced by LLM-VeriOpt improve code behavior under correctness constraints? **A2:** LLM-VeriOpt produces *different correct* output in 90% of cases. It improves performance in 84% of cases, instruction count in 86% of cases, and code size in 80% of cases, all with verifiably correct output. While the foundation model produces 16.4% *different correct* cases, it only *improves* performance in 1.2% of cases relative to `-O0`.

**RQ3 (Competitiveness):** How does LLM-VeriOpt compare with both LLM-based compilers and the traditional `LLVM -instcombine` pass? **A3:** LLM-VeriOpt outperforms standard supervised fine-tuned models that are over 10× larger in parameter size, in both correctness and performance improvement. Its performance improvement is comparable to

[6]This is an approximation of latency, which would fail with more complex transformations than peephole, such as loop unrolling where static instruction count would grow, or where different instructions in the real pipeline would conditionally overlap in ways that would affect the best result (which could be fixed by switching to pipeline-aware latency measurements such as via LLVM-mca. Still, since we only try to approximate peephole-style optimizations, the output produces good results in practice (see section V-E), and other peephole techniques e.g. Souper [51] use even simpler latency metrics.

TABLE I: Alive2 verification results of baseline Qwen-3B.

| Category | Count | Proportion (%) |
|---|---|---|
| Correct (Alive2 verified) | 3,210 | 73.2 |
| *– Copy of input (no optimization)* | *(2,490)* | *(56.8)* |
| Semantic Error (Not Equivalent) | 185 | 4.2 |
| Syntax Error (Invalid IR) | 927 | 21.1 |
| Inconclusive | 64 | 1.5 |

LLVM's handwritten `-instcombine` pass (2.30× speedup vs 2.39×) and produces superior results in 20.1% of cases, demonstrating LLM-VERIOPT's ability to produce emergent optimizations not included in the `-instcombine`-generated training set. With a fallback for when the LLM-VERIOPT output is worse, the net performance improvement is 17% relative to `-instcombine` alone.

**RQ4 (Ablation):** What is the contribution of design choices behind the hierarchy of models, including supervised warm-up and diagnostic-augmented sample-based prompt augmentation presented in section III-C? **A4:** Each of the four progressive sub-models gives critical progressive contributions towards all of latency, instruction count, binary size, and correctness.

### *A. [RQ1] Characterizing the Peephole Optimization Capability of Foundation Models*

We conducted an experiment with baseline Qwen-3B (i.e., without our verification-guided GRPO) to evaluate its correctness in generating IR under peephole optimization. Correctness was assessed using Alive2 to verify semantic equivalence.

In these initial experiments, we directly applied the generic prompt shown in Figure 1. For a small subset of test cases, we observed that the model frequently failed to generate outputs in the required format (with the `<answer>` tag) and produced IR that was largely syntactically invalid. To obtain meaningful statistics, we refined the prompting strategy by introducing one-shot learning: providing a sample pair of input IR and its optimized output. This adjustment enabled the model to consistently follow the required format and slightly improved correctness. The results are summarized in Table I.

Qwen-3B produced Alive2-verified IR for 73.2% of the cases. However, a large fraction of these (56.8%) were trivial copies of the input IR with no optimization applied[7]. The actual rate of semantically correct *different* (thus potentially useful) optimizations was therefore much lower at 16.4%. Additionally, 21.1% of the outputs were syntactically invalid and could not be parsed as legal IR, 4.2% were identified as semantically incorrect, and 1.5% were inconclusive.

### *B. [RQ2] Evaluating the Effectiveness of* LLM-VERIOPT

Table II summarizes the Alive2 verification results of the LLM-VERIOPT models. Excluding trivial copies of the input, MODEL-CORRECTNESS successfully improves 88.2% of samples, which is over **5.4× higher** than Qwen-3B (16.4%). We further evaluate the MODEL-LATENCY to examine whether incremental learning compromises IR correctness, and find that its correctness remains stable.

[7]While these answers are correct, they are no more useful than the prompt "please return the input as the output", which would have near-100% accuracy but no optimization capability. Sometimes the output of peephole optimization should be identical to the input, as no further optimization can occur – but for `-instcombine` this was not true for any of the samples in our test set.

TABLE II: Alive2 verification results of Qwen-3B augmented by LLM-VERIOPT models.

| Category | Model-Correctness Count | Model-Correctness % | Model-Latency Count | Model-Latency % |
|---|---|---|---|---|
| Correct | 3,926 | 89.5 | 3,940 | 89.9 |
| *– Copy of input (no opt.)* | *(59)* | *(1.4)* | *(67)* | *(1.5)* |
| Semantic Error | 227 | 5.2 | 237 | 5.4 |
| Syntax Error | 161 | 3.7 | 132 | 3.0 |
| Inconclusive | 72 | 1.6 | 66 | 1.5 |

TABLE III: Per-sample outcome counts vs. LLVM -O0 (smaller = better). The last column reports the mean relative change against -O0 (negative = improvement).

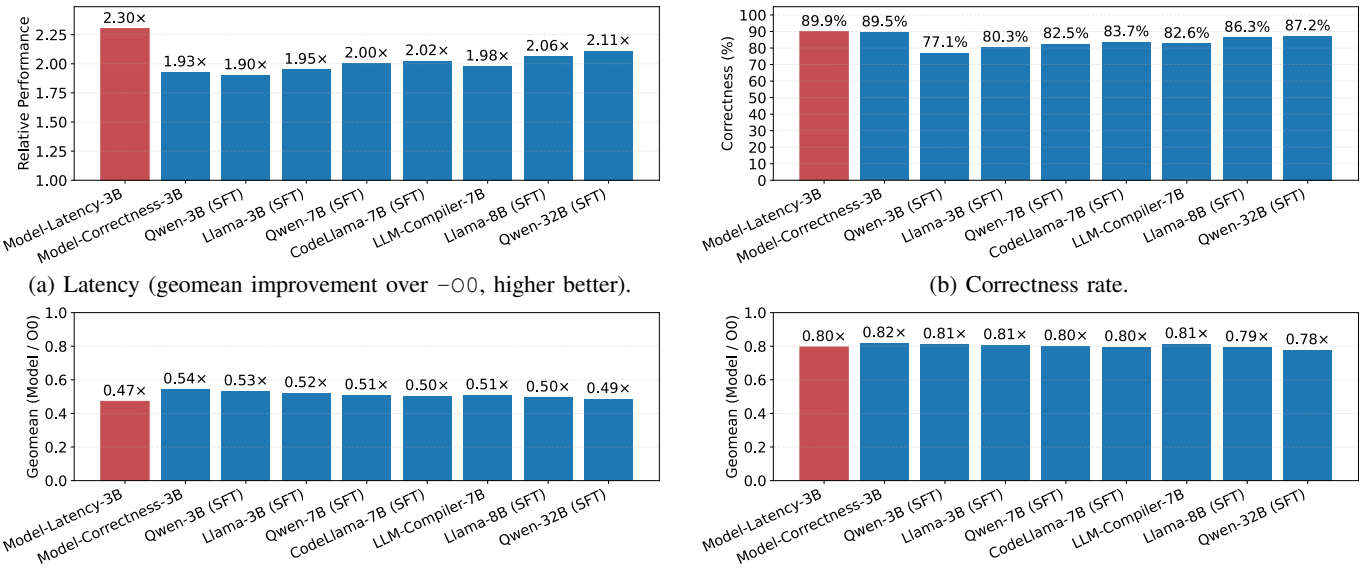
| Metric | Model | Better | Worse | Tie | Total | Mean Δ vs -O0 |
|---|---|---|---|---|---|---|
| Latency | Latency | **3696** | **0** | **690** | 4386 | **−50.68%** |
| | Correctness | 3556 | 1 | 829 | 4386 | −38.22% |
| | Qwen-3B | 53 | 40 | 4293 | 4386 | −0.19% |
| Size | Latency | **3528** | 105 | **753** | **4386** | **−17.37%** |
| | Correctness | 3416 | 50 | 920 | 4386 | −14.25% |
| | Qwen-3B | 64 | 31 | 4291 | 4386 | −0.15% |
| ICount | Latency | **3748** | **0** | **638** | **4386** | **−45.64%** |
| | Correctness | 3630 | 0 | 756 | 4386 | −33.70% |
| | Qwen-3B | 62 | 32 | 4292 | 4386 | −0.12% |

We further evaluate the models in terms of Instruction Count, Latency, and Binary Size. For outputs that pass Alive2 verification, we directly compute performance metrics; if verification fails, we fall back to the LLVM -O0 version.

The results are shown in Table III. MODEL-LATENCY achieves a substantial reduction in latency: relative to LLVM -O0, the improvement is also significant. By contrast the foundation model rarely generates useful code: while the *different accurate* (thus potentially useful) rate is 16.4% as discussed in section V-A is already low, the proportion of code that is actually improved (rather than different but the same latency or higher) is just 1.2%, and rarely is any function improved significantly, resulting in only a 0.19% mean improvement in latency. Instruction count is also greatly reduced, with MODEL-LATENCY achieving an average decrease of 45.6% relative to LLVM -O0, and an average reduction of 17.4% in binary size relative to LLVM -O0.

### *C. [RQ3] Evaluation against LLM/Compiler Baselines*

**Versus LLM-based compilers:** We compare LLM-VERIOPT against a range of LLM-based compilers, covering both supervised fine-tuning (SFT) baselines and state-of-the-art LLMs. Since small- and medium-scale LLMs exhibit limited capability in optimizing LLVM IR out-of-the-box, we perform SFT to ensure a fair comparison. For all SFT baselines, we adopt the generic prompt template illustrated in Figure 1 and train on the same dataset as LLM-VERIOPT until convergence, selecting the best checkpoint for evaluation. We further compare




(a) Latency (geomean improvement over `-O0`, higher better).

(b) Correctness rate.

(c) Instruction count (geomean versus `-O0`, lower better).

(d) Binary size (geomean versus `-O0`, lower better).

Fig. 5: Comparative performance of LLM-VERIOPT and baseline LLM-based compilers, presented in parameter-size order (in billions). Improvements for latency, binary size, and instruction count are reported as geomean relative to LLVM `-O0`. Correctness is measured as the percentage of semantically equivalent outputs verified by Alive2.

against a state-of-the-art model without task-specific fine-tuning: Cummins et al.'s LLM-Compiler-7B [4].

As shown in fig. 5, MODEL-LATENCY achieves the best results on latency, inst-count, and accuracy, even outperforming larger models such as Qwen-32B[8]. Larger models generally perform better, yet the small-scale MODEL-LATENCY bucks this trend, surpassing all baselines across most metrics.

**Versus LLVM:** In Figure 6, we compare LLM-VERIOPT's MODEL-LATENCY and LLVM's `-instcombine` across *Latency*, *ICount*, and *Binary Size*. Panels (a) and (b) show that LLM-VERIOPT achieves improvements over `-O0` that are broadly similar to those of `-instcombine`, indicating that the model has successfully learned to match the optimization capability of a hand-engineered pass. Panel (c) further compares LLM-VERIOPT directly with `-instcombine`: for latency - the primary optimization target - LLM-VERIOPT outperforms `-instcombine` in **20.1%** of functions, underperforms in **22.6%**, and ties in **57.3%**, with similar patterns observed for the other two metrics. Crucially, the cases where LLM-VERIOPT surpasses `-instcombine` cannot be attributed to memorization: the model was trained using labels generated by `-instcombine`, thus these additional gains are likely enabled by reinforcement learning.

MODEL-LATENCY achieves a geomean 2.30× speedup versus `-O0`, highly comparable with `-instcombine`'s 2.39× speedup. With a fallback, using model-generated IR only when it outperforms `-instcombine`, we achieve significant geomean improvements: latency 17% gain, instruction count 13.9%, and binary size 2.1%.

[8]Qwen-32B attains the best improvement in size: unsurprising since LLM-VERIOPT's reinforcement learning only implicitly optimizes for it via latency correlation (section III-C3); other reward functions yield different outcomes.

### *D. [RQ4] Analyzing the Contribution of Training Strategies and Prompt Augmentation*

We ablate the effect of incorporating *Alive2*-derived diagnostic information into both training and prompting. Specifically, we compare the four models presented in section III: (i) MODEL ZERO (GRPO-trained without Alive2 feedback, section III-C1), (ii) the WARM-UP MODEL, section III-C2, trained by supervised fine-tuning from MODEL ZERO's diagnostic-augmented samples, and (iii) MODEL-CORRECTNESS (also section III-C2), using GRPO to progressively improve Alive2 feedback emulation and code correctness combined, and the final LLM-VERIOPT mechanism, MODEL-LATENCY, improving MODEL-CORRECTNESS by incremental learning to induce a latency-oriented reward without losing correctness (section III-C3. As shown in fig. 7, each stage adds critical improvements. MODEL ZERO alone is already effective compared with the base Qwen-3B, which table III shows gains less than 0.2% on latency, instruction count and binary size[9]. The WARM-UP MODEL boosts speedup by boosting accuracy, meaning more code gets successfully modified and improved, by gaining a rudimentary ability to predict and fix bugs. MODEL-CORRECTNESS takes this a step further by co-optimizing bug-finding and code-generating capability via GRPO. Finally, MODEL-LATENCY builds on this further by retargeting the code to improve latency rather than just mimic `-instcombine`. In fact, MODEL-LATENCY also gets a better accuracy than MODEL-CORRECTNESS despite additional latency optimization criteria:

[9]This is despite MODEL-ZERO's semantic accuracy (50.1%) being lower than Qwen's 73.2% meaning it leaves more code at its unprocessed baseline performance, but as we discuss in section V-A, this is because the Qwen baseline makes no meaningful attempt to actually emulate instcombine or optimize the code, instead just repeating the input in the majority of cases it returns a valid answer, being vacuously correct.

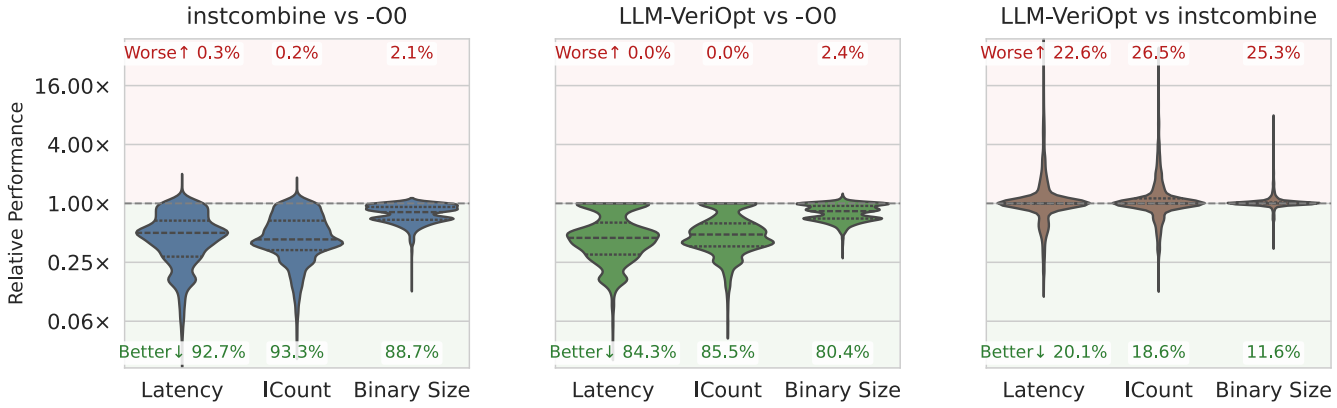



Fig. 6: Pairwise distributions of optimized IR against baselines across *Latency*, *Instruction Count* and *Binary Size*.

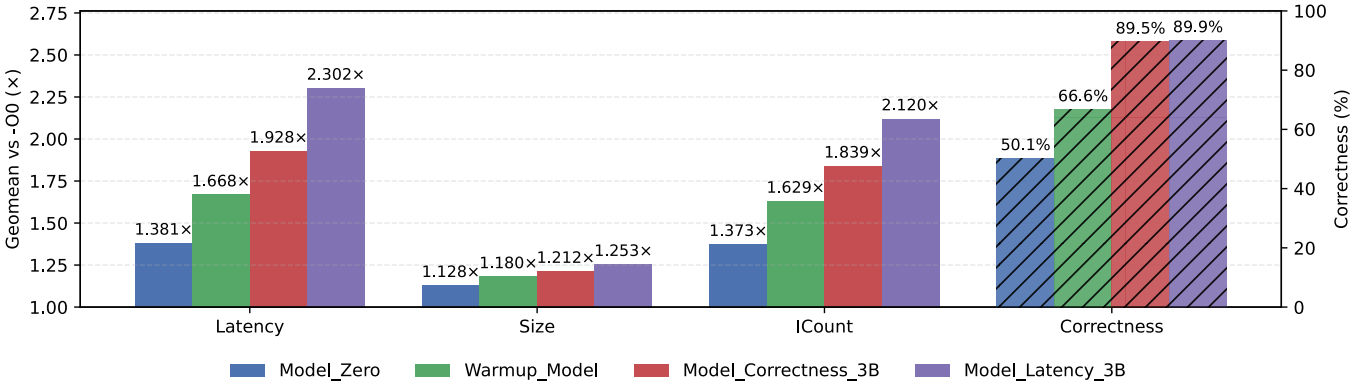



Fig. 7: Ablation study: geomean improvements (vs `-O0`, higher is better) for Latency/ICount/Binary Size (left axis) and Correctness (%) (right axis). We compare four variants: MODEL ZERO, a GRPO-trained 3B model with generic prompts; the WARM-UP MODEL, supervised fine-tuned on augmented prompts; the MODEL-CORRECTNESS; and the MODEL-LATENCY.

we attribute this to the fact that MODEL-LATENCY does not have to waste its very finite parameter budget on producing actual error messages because they are no longer needed once MODEL-CORRECTNESS has finished using them for its GRPO, allowing MODEL-LATENCY to retain them only implicitly to preserve its reasoning capability.

### E. Code Examples

LLM-VeriOpt sometimes reduces complex control/data flow into a single return value where InstCombine does not (fig. 8). Many emergent optimizations involve complex value propagation, including fig. 9. Figure 10 appears to be learning elements of other LLVM passes (simplifycfg, we also saw mem2reg-like behavior) from the reward function and Alive-correctness alone, without explicit finetuning training data and despite base models failing to do so. VeriOpt does not spot all InstCombine patterns, perhaps from training-set limitations and from too few model parameters to fully represent InstCombine: fig. 11 misses a truncation, and in fig. 12 InstCombine fully precalculates unlike VeriOpt, likely from GRPO discouraging VeriOpt from attempting arithmetic LLMs are poor at.

*InstCombine:*

```
%struct.S = type { i32, i32 }
define dso_local i64 @get_d() #0 {
  %1 = alloca i64, align 8
  %tmpcast = bitcast i64* %1 to
      %struct.S*
  %2 = bitcast i64* %1 to i32*
  store i32 0, i32* %2, align 8
  %3 = getelementptr inbounds
      %struct.S, %struct.S*
      %tmpcast, i64 0, i32 1
  store i32 0, i32* %3, align 4
  %4 = load i64, i64* %1, align 8
  ret i64 %4
}
```

*LLM-VeriOpt:*

```
define dso_local i64
    @get_d() #0 {
  ret i64 0
}
```

Fig. 8: Simplification to 0.

## VI. DISCUSSION

**On the role of GRPO without explicit reasoning.** Although GRPO is often intended to encourage models to autonomously generate an explicit chain of thought, in our experimental setting, this mechanism did not function as expected. Instead, we primarily relied on manually constructed reasoning chains—namely, *augmented prompts* that embed *Alive2* diagnostic information—serving as "externally provided chains of thought" rather than model-generated ones. The underlying cause lies in model scale: our 3B-parameter model lacks sufficient prior knowledge of LLVM IR to produce coherent and meaningful reasoning steps. During training, the chain of thought generated autonomously by the model is typically noisy and uninformative, and thus contributed little to improving optimization capability. We hypothesize, however, that when applied to larger foundation models with richer IR

*InstCombine:*

```
define dso_local i64 @f28(i64
      noundef %0, i64 noundef
      %1) #1 {
  %3 = alloca i64, align 8
  %4 = add i64 %0, %1
  store i64 %4, i64* %3,
       align 8
  %.not = icmp ugt i64 %4, %0
  br i1 %.not, label %6,
       label %5
5: call void @foo(i32 noundef
      0) #2
  br label %6
6: %7 = load i64, i64* %3,
     align 8
  ret i64 %7
}
```

*LLM-VeriOpt:*

```
define dso_local i64 @f28(i64
      noundef %0, i64 noundef
      %1) #1 {
  %3 = add i64 %0, %1
  %4 = icmp uge i64 %0, %3
  br i1 %4, label %12, label
       %13
12: call void @foo(i32 0)
  br label %13
13: ret i64 %3
}
```

Fig. 9: VeriOpt removes redundant allocas, stores and loads.

*InstCombine:*

```
define dso_local i32 @opt_u1(
     i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2,
       align 4
  %3 = icmp ult i32 %0, 10
  br i1 %3, label %4, label
       %5
4: br label %10
5: %6 = load i32, i32* %2,
     align 4
  %7 = add i32 %6, -12
  %8 = lshr i32 %7, 2
  %9 = add nuw nsw i32 %8, 3
  br label %10
10: %storemerge = phi i32 [
     %9, %5 ], [ 0, %4 ]
  ret i32 %storemerge
}
```

*LLM-VeriOpt:*

```
define dso_local i32 @opt_u1(
     i32 noundef %0) #0 {
  %2 = icmp ult i32 %0, 10
  br i1 %2, label %8, label
       %9
8: ret i32 0
9: %10 = sub i32 %0, 12
  %11 = udiv i32 %10, 4
  %12 = add i32 %11, 3
  ret i32 %12
}
```

Fig. 10: Emergent learning of simplifycfg-style behavior.

semantic knowledge (e.g., a 32B-parameter model), GRPO could potentially elicit more coherent self-generated reasoning chains, which may lead to stronger optimization performance.

**Limitations of Alive2.** We found a couple examples in the test-set where Alive2 gets the answer wrong, likely because of Alive2's well-documented limitations around loop analysis [15]. This was very rare seemingly because InstCombine does not do complex loop transformation, nor does our reward function incentivize it appearing emergently, meaning changes appear within a few bounded unrolls – but Alive2 currently cannot guarantee correctness even then, and this is likely to cause practical impediments with loop-level analyses. This is not a theoretical impediment – with support for loop induction, or constraints over loop-format changes to make the problem tractable, we expect soundness would improve.

## VII. Related work

In recent years, there has been a growing interest in leveraging Large Language Models (LLMs) for tasks involving source code generation. Models such as Copilot [52], Codex [53], TransCoder [54], CodeBERT [55], Code Llama [56], StarCoder [57], [58], Magicoder [59] and DeepSeek-Coder [60] have significantly advanced this field. These models support developers with tasks like code completion, generation, and translation across multiple programming languages.

*InstCombine:*

```
define dso_local i32 @f8(i64
     noundef %0) #0 {
  %2 = lshr i64 %0, 61
  %3 = trunc i64 %2 to i32
  %4 = add nuw nsw i32 %3, 1
  ret i32 %4
}
```

*LLM-VeriOpt:*

```
define dso_local i32 @f8(i64
      noundef %0) #0 {
  %2 = lshr i64 %0, 61
  %3 = trunc i64 %2 to i16
  %4 = sext i16 %3 to i32
  %5 = add nsw i32 %4, 1
  %6 = trunc i32 %5 to i16
  %7 = sext i16 %6 to i32
  ret i32 %7
}
```

Fig. 11: InstCombine spots a superior simplification.

*InstCombine:*

```
define dso_local i32
     @aqua_baldo() #0 {
  ret i32 -159
}
```

*LLM-VeriOpt:*

```
define dso_local i32
     @aqua_baldo() #0 {
  %1 = srem i32 160, 11
  %2 = sub nsw i32 %1, 2
  %3 = add nsw i32 6, %2
  %4 = srem i32 11, %3
  %5 = sub nsw i32 1, 160
  %6 = srem i32 %2, %4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```

Fig. 12: InstCombine fully precalculates.

Fewer models operate at the compiler level, particularly with code generation and compiler optimization. Recent studies have focused on traditional machine-learning methods for compiler optimization [44], [61]–[67]. Neural machine translation techiques have been employed to transform code between different representations, previous examples include compiling C to X86 assembly [8] and decompiling assembly language to C [68], [69]. These works utilized smaller models or other deep learning methods. There are a few works related to using LLM at the compiler level. Examples include using large models for decompilers [70]–[72], LLVM-IR passes prediction with IR optimization [3], and fuzzing tests [73], [74].

## VIII. Conclusion

In this paper, we present LLM-VeriOpt, a framework for producing high-quality, accurate, and performant optimization passes via LLMs. LLM-VeriOpt takes classification data from a compiler pass it is trying to emulate, observes the mistakes it makes in doing so, learns to diagnose these mistakes, and ultimately corrects them. Through a hierarchy of models, LLM-VeriOpt is able to transform the Qwen-3B LLM from an 0.2% speedup to over 2.3×, by both vastly improving the coverage of code by generating verifiably correct optimizations, and also by improving the optimization of the code it covers by generating high-quality transformations. We believe the three-stage framework we provide here has significant potential in facilitating the development of ever more ambitious LLM compiler passes, which, when scaled up to larger models and more ambitious training sets will allow consistent and large improvements over handwritten compiler passes, as well as generating code that is easy to verify with models such as Alive2, sidestepping traditional LLM correctness barriers completely.

## ARTIFACT APPENDIX

*Artifact Abstract*

This artifact is designed to run on a modern Linux environment. It provides all trained models, datasets, inference scripts, configuration files, and reproduction materials required to replicate the results presented in the paper. It includes: (1) all trained SFT, GRPO LoRA models; (2) all test datasets; (3) a unified inference pipeline; (4) configuration files covering every model variant; (5) scripts to reproduce the figures used in the paper; (6) author-provided reference outputs for verification; and (7) a complete summary table that aggregates each model's outputs across the full test set, including IR size, latency, instruction count, and correctness metrics. reported in the paper are generated directly from this summary table.

The artifact supports both full test-set evaluation and lightweight sampling-based evaluation. **We strongly recommend using the sampling script** (`run_inference_demo.sh`) for practical evaluation on commodity hardware. The full artifact, including models, scripts, datasets, and reproducibility materials, is publicly available on GitHub and Zenodo at https://github.com/carrotProgrammer/llmveriopt-AE and https://doi.org/10.5281/zenodo.17625555

*1. Artifact Check List*

- **Models:** All LoRA adapters included.
- **Datasets:** All test datasets included.
- **Output data:** Author-produced reference results in `reference_results/`.
- **Scripts:** Sampling evaluation, full evaluation, and figure-generation scripts provided.
- **Hardware Requirements:**
  - Recommended: Nvidia GPU $\geq$ 32 GB for 7B/8B/32B models.
  - Minimum: Nvidia GPU $\geq$ 16 GB for 3B models.
  - CPU execution is supported (with fallback if no Nvidia GPU is detected), but extremely slow (may take days).
- **Software Requirements:** Linux, Python 3.10, PyTorch, Transformers, PEFT, datasets, YAML.
- **Estimated Runtime:** (The following runtime estimates are based on measurements using an NVIDIA RTX 3090ti GPU.)
  - Sampling evaluation: 1hr. (10-16hr on CPU with default LIMIT=32)
  - Full evaluation for model_latency_3b: 9–12h.
  - Full 3B/7B/8B/32B evaluation with large-GPU support: multiple days.
- **Archived:** https://doi.org/10.5281/zenodo.17625555

*2. Dependencies*

All dependencies can be installed using:

```
pip install -r requirements.txt
```

Key packages include: `torch`, `transformers`, `peft`, `datasets`, `pyyaml`. All experiments were tested using the package versions listed in `requirements.txt`, running on Ubuntu 24.04. Other compatible versions may work but have not been systematically evaluated.

Gated models provided by Meta (e.g., the Llama model family) require HuggingFace authentication Evaluators must first request access on the corresponding model card page, then generate a personal access token on HuggingFace, and finally authenticate locally using:

```
huggingface-cli login
```

By default, base models are accessed directly from HuggingFace. If network access to HuggingFace is unavailable, evaluators may manually download the required models in advance and update the `base_model` fields in all configuration files to point to the corresponding local paths.

*3. Installation and Directory Structure*

The artifact consists of two components: (1) the main repository hosted on GitHub, and (2) the evaluation dataset hosted on Zenodo. Users must obtain both before running.

*3.1 Obtaining the Artifact:*

*a) Main repository:* The primary artifact repository, including trained models, inference scripts, configuration files, and figure-reproduction scripts, is available at:

https://github.com/carrotProgrammer/llmveriopt-AE

Users may clone it with:

```
git clone https://github.com/carrotProgrammer/
    llmveriopt-AE
cd llmveriopt-AE
```

*b) Evaluation dataset (Zenodo).:* The evaluation dataset required by the artifact is provided via Zenodo:

https://doi.org/10.5281/zenodo.17625556

After downloading `llmveriopt-datasets.zip`, extract it and place the resulting `dataset/` directory directly under the artifact root:

```
unzip llmveriopt-datasets.zip

llmveriopt-AE/
'-- dataset/
    '-- <dataset files>
```

This directory contains all test IR programs, latency benchmarks, and reference outputs used by the evaluation pipeline.

*3.2 Directory Structure:* After cloning the repository and placing the dataset, the directory layout should be:

```
llmveriopt-AE/
|-- models/     # LoRA adapters and baseline model
    references
|-- dataset/    # Evaluation datasets (from Zenodo)
|-- inference/
|   |-- run_inference_demo.sh
|   |-- run_inference_all.sh
|   |-- run_model_latency.sh
|   |-- output/     # Evaluation logs and generated
    results
|   |-- tools/    # Alive2 bindings and prebuilt
    LLVM shared libraries
|   '-- configs/*.yaml
|-- reproduce_figures/
|   |-- summary_table/
|   '-- reproduce_figures.sh
|-- requirements.txt
'-- README.md
```

All LLVM shared libraries needed for Alive2 (e.g., `libLLVM.so`) are included in:

`inference/tools/llvm-project/build/lib/`

No additional compilation steps are required.

*3.3 Dependencies:* The artifact requires Python 3.10 or newer. Install all Python dependencies via:

```
pip install -r requirements.txt
```

Alive2 further requires the Z3 SMT solver. On Debian/Ubuntu:

```
sudo apt update
sudo apt install z3 libz3-dev
```

If users wish to evaluate gated models (e.g., Llama-family checkpoints), HuggingFace authentication is required:

```
huggingface-cli login
```

*Hardware Notes:* While CPU execution is supported, we strongly recommend running on an NVIDIA GPU with at least 16 GB of memory. If no GPU is detected, the scripts automatically fall back to CPU mode.

Users may explicitly force CPU execution by clearing the CUDA device list:

```
export CUDA_VISIBLE_DEVICES=""
```

or specify a particular GPU index if multiple devices exist.

## 4. Evaluation

*4.1 Quick Functional Test (Recommended):* A lightweight sampling-based evaluation is provided to verify that model inference, IR generation, and Alive2 verification all function correctly. This mode uses a small subset of the dataset and typically completes within minutes on commodity hardware.

To run the quick evaluation:

```
cd inference
chmod +x run_inference_demo.sh
./run_inference_demo.sh
```

Results are written to:

```
inference/output/new_result/
```

Each evaluated model generates:

```
<model_name>/
|-- results.csv        # Per-function evaluation
    records
`-- metrics.json       # Summary and correctness
    metrics
```

A summary figure (`summary.png`) is produced and may be compared against the reference file:

```
inference/output/reference_results/summary.png
```

This procedure validates that the full evaluation pipeline is functioning correctly. **We strongly encourage reviewers to begin with this mode** prior to running the full suite.

*4.2 Full Reproduction (Not Recommended on Small GPUs):* To reproduce all results:

```
cd inference
chmod +x run_inference_all.sh
./run_inference_all.sh
```

This evaluates all models (3B/7B/8B/32B) on the full test sets (up to 4386 samples). **Warning:** This requires an Nvidia GPU with $\geq$32 GB memory. On smaller devices, inference fail due to out of VRAM.

*Note: this script performs inference only. It does not generate metrics or summary figures.*

*4.3 Final Model Evaluation (model_latency) - only:* The script `run_model_latency.sh` runs the full evaluation for our final model, `model_latency`, which is the primary model used to produce the main results reported in the paper. This model achieves the best overall performance among all evaluated variants, and is a subset of the experiments implemented for the full reproduction. Since it only evaluates a 3B model, memory requirements are 16 GB instead of 32 GB.

```
cd inference
chmod +x run_model_latency.sh
./run_model_latency.sh
```

This script runs the full latency evaluation over the entire test set and is computationally expensive. On an Nvidia RTX 3090 GPU, the expected runtime is approximately 9–12 hours. Running the script on smaller GPUs may lead to out-of-memory failures.

*Note: This script reproduces the inference output for the primary model (Model_Latency) used in the paper. It does not compute metrics/generate plots.*

*4.4 Experiment Modification:* All three scripts above share the same logic; they differ only in the model list and the LIMIT settings. `run_inference_all.sh` runs all models (Meta + Qwen) on the full test set, `run_model_latency.sh` runs only the `model_latency_3b` configuration on the full test set, and `run_inference_demo.sh` runs the 3B models with a smaller, user-configurable sample limit. Reviewers may adjust both the model list and the LIMIT directly in the relevant `.sh` files.

## 5. Expected Results

Running the sampling script produces:

- CSV outputs under `inference/output/new_result/<model>/results.csv`.
- Deterministic IR generation patterns for sampled tests.

Reviewers should compare:

`inference/output/new_result/summary.png`

against the reference version:

`inference/output/reference_results/summary.png`

to confirm reproducibility of the evaluation pipeline.

All IR outputs, Alive2 verification logs, and detailed metrics are stored under each `<model_name>` directory.

## 6. Notes for Evaluators

The artifact is compatible with any modern Linux system. However, hardware with insufficient GPU memory will encounter out-of-memory errors when running larger models. We recommend only running on CPUs (as described above) and only running the functional tests if so, to limit compute time.

All scripts are deterministic because model generation uses greedy decoding.

## REFERENCES


[1] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 697–709. [Online]. Available: https://doi.org/10.1145/3503222.3507764

[2] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "Compilergym: robust, performant compiler optimization environments for ai research," in *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '22. IEEE Press, 2022, p. 92–105. [Online]. Available: https://doi.org/10.1109/CGO53902.2022.9741258

[3] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large language models for compiler optimization," 2023. [Online]. Available: https://arxiv.org/abs/2309.07062

[4] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Llm compiler: Foundation language models for compiler optimization," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 141–153. [Online]. Available: https://doi.org/10.1145/3708493.3712691

[5] S. Tang, C. Priebe, R. Mahapatra, L. Qin, and H. Esmaeilzadeh, "Compiler optimization via llm reasoning for efficient model serving," 2025. [Online]. Available: https://arxiv.org/abs/2506.01374

[6] X. Fang and L. Mukhanov, "Towards llm-based optimization compilers. can llms learn how to apply a single peephole optimization? reasoning is all llms need!" 2024. [Online]. Available: https://arxiv.org/abs/2412.12163

[7] A. Wei, T. Suresh, H. Tan, Y. Xu, G. Singh, K. Wang, and A. Aiken, "Supercoder: Assembly program superoptimization with large language models," 2025. [Online]. Available: https://arxiv.org/abs/2505.11480

[8] J. Armengol-Estapé and M. F. P. O'Boyle, "Learning c to x86 translation: An experiment in neural compilation," NeurIPS 2021 AIPLANS Workshop, 2021. [Online]. Available: https://arxiv.org/abs/2108.07639

[9] Z. Gao, H. Wang, Y. Wang, and C. Zhang, "Vic: Virtual compiler is all you need for assembly code search," 2024. [Online]. Available: https://arxiv.org/abs/2408.06385

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[11] H. Face, "Qwen2.5-coder-7b: Code-specific qwen large language models," https://huggingface.co/Qwen/Qwen2.5-Coder-7B, 2024, accessed: 2025-09-10.

[12] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Meta large language model compiler: Foundation models of compiler optimization," 2024. [Online]. Available: https://arxiv.org/abs/2407.02524

[13] J. Taneja, A. Laird, C. Yan, M. Musuvathi, and S. K. Lahiri, "Llm-vectorizer: Llm-based verified loop vectorizer," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 137–149. [Online]. Available: https://doi.org/10.1145/3696443.3708929

[14] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3703155

[15] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: bounded translation validation for llvm," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 65–79. [Online]. Available: https://doi.org/10.1145/3453483.3454030

[16] N. Lambert, J. Morrison, V. Pyatkin, S. Huang, H. Ivison, F. Brahman, L. J. V. Miranda, A. Liu, N. Dziri, S. Lyu, Y. Gu, S. Malik, V. Graf, J. D. Hwang, J. Yang, R. L. Bras, O. Tafjord, C. Wilhelm, L. Soldaini, N. A. Smith, Y. Wang, P. Dasigi, and H. Hajishirzi, "Tulu 3: Pushing frontiers in open language model post-training," 2025. [Online]. Available: https://arxiv.org/abs/2411.15124

[17] X. Wen, Z. Liu, S. Zheng, Z. Xu, S. Ye, Z. Wu, X. Liang, Y. Wang, J. Li, Z. Miao, J. Bian, and M. Yang, "Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms," 2025. [Online]. Available: https://arxiv.org/abs/2506.14245

[18] Y. Mroueh, "Reinforcement learning with verifiable rewards: Grpo's effective loss, dynamics, and success amplification," 2025. [Online]. Available: https://arxiv.org/abs/2503.06639

[19] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," 2024. [Online]. Available: https://arxiv.org/abs/2402.03300

[20] H. Face, "Qwen2.5-3b-instruct:an instruction-tuned language model," https://huggingface.co/Qwen/Qwen2.5-3B-Instruct, 2023, accessed: 2025-09-10.

[21] LLVM, "Instructioncombining.cpp source file," Accessed: Jul. 7, 2024, Jul 2024, [Online]. Available: https://llvm.org/doxygen/InstructionCombining_8cpp_source.html.

[22] LLVM Project, "Llvm test suite," https://github.com/llvm/llvm-test-suite, 2025, accessed: 2025-09-03.

[23] Free Software Foundation, *Installing GCC: Testing*, 2025, Note: GCC Testsuite is distributed as part of the GCC source tree. Accessed: 2025-09-03. [Online]. Available: https://gcc.gnu.org/install/test.html

[24] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. [Online]. Available: https://dl.acm.org/doi/10.5555/977395.977673

[25] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Academic Press/Morgan Kaufmann, 1997.

[26] C. N. Fischer, R. K. Cytron, and R. J. L. Jr., *Crafting a Compiler*. Boston, MA, USA: Addison-Wesley, 2010.

[27] N. Popov, "How single-iteration instcombine improves llvm compile time," *Red Hat Developer*, Dec. 2023, accessed: 2025-09-09. [Online]. Available: https://developers.redhat.com/articles/2023/12/07/how-single-iteration-instcombine-improves-llvm-compile-time

[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[29] S. Xu, W. Fu, J. Gao, W. Ye, W. Liu, Z. Mei, G. Wang, C. Yu, and Y. Wu, "Is DPO superior to PPO for LLM alignment? A comprehensive study," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 54 983–54 998. [Online]. Available: https://proceedings.mlr.press/v235/xu24h.html

[30] Y. Wang and F. Xie, "Enhancing translation validation of compiler transformations with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2401.16797

[31] J. Lee, D. Kim, C.-K. Hur, and N. P. Lopes, "An smt encoding of llvm's memory model for bounded translation validation," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 752–776. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_35

[32] "Llvm qualification wg sync-ups meeting minutes," 2025. [Online]. Available: https://discourse.llvm.org/t/llvm-qualification-wg-sync-ups-meeting-minutes/87148

[33] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[34] M. Pezeshki, O. Kaba, Y. Bengio, A. C. Courville, D. Precup, and G. Lajoie, "Gradient starvation: A learning proclivity in neural networks," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 1256–1272. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/0987b8b338d6c90bbedd8631bc499221-Paper.pdf

[35] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[36] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[37] Nikita Popov, "Issue #151303 on llvm/llvm-project: [instcombine] incorrect fabs + nsz fold," GitHub issues, Jul. 2025, https://github.com/llvm/llvm-project/issues/151303.

[38] Nuno Lopes, "Issue #156435 on llvm/llvm-project: Instcombine/lowerobjectsizecall introducing an assume about allocation sizes," GitHub issues, 2025, https://github.com/llvm/llvm-project/issues/156435.

[39] Yingwei Zheng, "Issue #157238 on llvm/llvm-project: [instcombine] wrong folding of is.fpclass + minnum," GitHub issues, 2025, https://github.com/llvm/llvm-project/issues/157238.

[40] ——, "Issue #157254 on llvm/llvm-project: [instcombine] wrong folding of select + fpext," GitHub issues, 2025, https://github.com/llvm/llvm-project/issues/157254.

[41] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024. [Online]. Available: https://aclanthology.org/2024.tacl-1.9/

[42] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 67–80. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444788

[43] J. Armengol-Estapé, R. C. O. Rocha, J. Woodruff, P. Minervini, and M. O'Boyle, "Forklift: An extensible neural lifter," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=LWfDcI6txJ

[44] Z. Zheng, K. Wu, L. Cheng, L. Li, R. C. O. Rocha, T. Liu, W. Wei, J. Zeng, X. Zhang, and Y. Gao, "Vectrans: Enhancing compiler auto-vectorization through llm-assisted code transformations," 2025. [Online]. Available: https://arxiv.org/abs/2503.19449

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[46] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," 2020. [Online]. Available: https://arxiv.org/abs/1904.09751

[47] J. Hu, Y. Zhang, Q. Han, D. Jiang, X. Zhang, and H.-Y. Shum, "Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model," 2025. [Online]. Available: https://arxiv.org/abs/2503.24290

[48] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, 1951.

[49] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2022. [Online]. Available: https://arxiv.org/abs/1312.6114

[50] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, W. Dai, T. Fan, G. Liu, L. Liu, X. Liu, H. Lin, Z. Lin, B. Ma, G. Sheng, Y. Tong, C. Zhang, M. Zhang, W. Zhang, H. Zhu, J. Zhu, J. Chen, J. Chen, C. Wang, H. Yu, Y. Song, X. Wei, H. Zhou, J. Liu, W.-Y. Ma, Y.-Q. Zhang, L. Yan, M. Qiao, Y. Wu, and M. Wang, "Dapo: An open-source llm reinforcement learning system at scale," 2025. [Online]. Available: https://arxiv.org/abs/2503.14476

[51] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," 2018. [Online]. Available: https://arxiv.org/abs/1711.04422

[52] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," 2023. [Online]. Available: https://arxiv.org/abs/2302.06590

[53] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[54] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020. [Online]. Available: https://arxiv.org/abs/2006.03511

[55] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[56] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[57] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[58] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024. [Online]. Available: https://arxiv.org/abs/2402.19173

[59] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," 2024. [Online]. Available: https://arxiv.org/abs/2312.02120

[60] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[61] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," 2021. [Online]. Available: https://arxiv.org/abs/2101.04808

[62] Z. Wang and M. O'Boyle, "Machine learning in compiler optimisation," 2018. [Online]. Available: https://arxiv.org/abs/1805.03441

[63] H. Leather and C. Cummins, "Machine learning in compilers: Past, present and future," *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–8, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221855481

[64] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. Leather, and Y. Tian, "Learning compiler pass orders using coreset and normalized value prediction," 2023. [Online]. Available: https://arxiv.org/abs/2301.05104

[65] A. Haj-Ali, Q. J. Huang, W. S. Moses, J. Xiang, K. Asanovic, J. Wawrzynek, and I. Stoica, "Autophase: Juggling HLS phase orderings in random forests with deep reinforcement learning," in *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, Eds. mlsys.org, 2020. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2020/hash/5b47430e24a5a1f9fe21f0e8eb814131-Abstract.html

[66] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams, "Using machine learning to focus iterative optimization," 04 2006, pp. 11 pp.–.

[67] A. F. Zanella, A. F. da Silva, and F. M. Quintão, "Yacos: a complete infrastructure to the design and exploration of code optimization sequences," in *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*, ser. SBLP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 56–63. [Online]. Available: https://doi.org/10.1145/3427081.3427089

[68] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. P. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," 2024. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444788

[69] I. Hosseini and B. Dolan-Gavitt, "Beyond the c: Retargetable decompilation using neural machine translation," in *Proceedings 2022 Workshop on Binary Analysis Research*, ser. BAR 2022. Internet Society, 2022. [Online]. Available: http://dx.doi.org/10.14722/bar.2022.23009

[70] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2403.05286

[71] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled c code with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2310.06530

[72] X. She, Y. Zhao, and H. Wang, "Wadec: Decompiling webassembly using large language model," 2024. [Online]. Available: https://arxiv.org/abs/2406.11346

[73] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," 2023. [Online]. Available: https://arxiv.org/abs/2212.14834

[74] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language models," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, p. 709–735, Oct. 2024. [Online]. Available: http://dx.doi.org/10.1145/3689736