Hydra: Multi-Objective Software Optimization using Large Language Models

Arjun Gupte
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, United States
guptea@purdue.edu

Ahmed Elmersawy
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, United States
aelmersa@purdue.edu

Andre Lee
Computer Science
Purdue University
West Lafayette, Indiana, United States
lee5897@purdue.edu

Stefan Maxim
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, United States
maxim@purdue.edu

***Abstract*—Automatic software system optimization can improve software speed, reduce operating costs, and save energy, but real deployments rarely optimize a single metric as developers must navigate trade-offs across performance, memory, readability, and other constraints that vary by workload and user preference. Traditional approaches rely on manual tuning and compiler heuristics, while recent Large Language Model (LLM)-based optimizers often struggle to represent multi-objective priorities in a controllable way across diverse settings. However, there remains a lack of methods that can robustly encode and adapt to varying multi-objective trade-offs within a single, unified optimization framework.**

Our system, Hydra, provides an end-to-end workflow that measures multiple objectives on candidate implementations and uses those measurements to fine-tune an LLM with Direct Preference Optimization (DPO), producing an optimizer that can adapt to different trade-offs. Given alternative implementations for the same task, the pipeline samples per-example objective weights, materializes a structured JSONL dataset, executes candidates to collect objective-specific measurements via profilers or host metrics, normalizes measurements into comparable reward components, and aggregates them into a weighted reward to label winner/loser pairs for DPO training. We validate the pipeline's end-to-end functionality on small-scale examples and describe ongoing work toward a systematic evaluation, along with the expected limitations and challenges of applying the approach to complex applications.

***Index Terms*—Software Optimization, Large Language Models, Preference Learning, Reinforcement Learning**

I. Introduction

Software optimization in real-world systems is inherently multi-objective: developers must balance trade-offs across runtime, memory usage, energy consumption, and other constraints that vary by workload and user preference. These trade-offs have direct consequences for reliability [1], user experience [2], energy consumption [3]–[5], and long-term sustainability [6]. Performance failures can lead to costly downtime and increased security risks [7]–[9], while data centers already account for 3-4% of electricity consumption in major regions [10]–[12]. Despite this, optimization is often deprioritized in practice due to competing demands such as rapid development and time-to-market [13], and is typically addressed only when bottlenecks become critical [14].

Existing approaches to Multi-Objective Software Optimization (MOSO) remain limited. Traditional techniques, including compiler optimizations [15], algorithmic improvements [16], and hardware-aware tuning [17], require significant manual effort and are not designed to flexibly adapt to changing trade-offs across objectives. Recent LLM-based systems demonstrate strong performance improvements, but lack explicit mechanisms for controllable preference learning across multiple objectives [18]. Other approaches incorporate preference learning via reinforcement learning or preference optimization (e.g., DPO/GRPO), but rely on online training loops that are computationally expensive [19]. As a result, existing methods either lack flexible multi-objective preference control or incur significant training overhead.

In modern computing systems, optimization targets are inherently multi-dimensional, spanning metrics such as runtime, memory usage, energy consumption, CPU cycles, and throughput, with priorities that vary across use cases. To address this, we frame MOSO as a preference learning problem and introduce **Hydra**, which leverages Direct Preference Optimization (DPO) to train an LLM to distinguish between better and worse implementations under varying objective trade-offs. By constructing preference pairs from measured performance across multiple metrics, the model learns to favor implementations that align with sampled objective weights, effectively moving toward desirable optimizations while avoiding inferior ones. This enables the system to understand multi-objective trade-offs and adapt its optimization behavior at inference time.

We evaluate Hydra on the *PIE* dataset, using diverse programming tasks with multiple candidate implementations to assess optimization quality across five metrics: runtime, energy consumption, memory usage, CPU cycles, and throughput. We also evaluate the performance of existing LLM-based opti-

mization baselines, including *Efficoder* and GPT-5.4. Finally, we present results demonstrating the feasibility of our pipeline along with a discussion of expected challenges and limitations.

II. Background

Reinforcement Learning (RL) for code generation and optimization presents an opportunity to address the limitations of non-AI-based and some alternative AI-based approaches. Prior RL works can be broadly categorized into the following two groups: 1) Direct RL and 2) Preference/Ranking Optimization.

A. *Direct Reinforcement Learning*

Direct RL methods leverage techniques including Actor-Critic Networks and Proximal Policy Optimization (PPO) to enhance code generation and optimization performance. For example, CodeRL uses the actor-critic framework and incorporates feedback from unit test case results [20]. A pre-trained LLM functions as the actor and generates code according to the problem specifications, while a separate critic network learns to predict the functional correctness of the code given reward signals from running unit tests. Although this approach helps reduce logical and syntactical errors in the optimized code, its reward signals are limited to functional correctness, and the training process is computationally expensive. PPOCoder similarly leverages the actor-critic paradigm, but introduces PPO with a clipped objective function to prevent significant policy updates and substantial deviations from the original pre-trained LLM [21]. This method provides denser reward signals compared to CodeRL by incorporating compiler feedback, syntactic matching scores, semantic matching scores, and a KL-Divergence penalty to facilitate the actor's policy updates. The use of PPO increases stability and generalization during training, but the use of a KL-Divergence penalty potentially hinders policy exploration.

B. *Preference Optimization*

Preference/Ranking Optimization methods rely heavily on Direct Preference Optimization (DPO) or variations of it to perform preference learning and avoid bottlenecks found in traditional RL. CodeDPO establishes preference relations between code snippets based on functional correctness and code performance [22]. These preference labels guide the LLM's code generation, increasing the probability of generating code aligned with the preferred priors. However, CodeDPO relies on a custom PageRank algorithm when generating preference labels, limiting its generalizability. Other approaches modify DPO to create RLPF (Reinforcement Learning with Performance Feedback) and Direct Performance Alignment (DPA) [23]. DPA treats code generation as a preference optimization problem, allowing the model to learn to differentiate between "fast" and "slow" code during training. Although this method obviates the need for an explicit reward model, which may suffer from complex reward shaping, its formulation limits it to the "fast" and "slow" pairwise code representations.

Metric	Units	Description
Functional Correctness	Pass/Fail	Code must compile and pass all tests; ensures semantic preservation.
Runtime	ms	Total execution time of the program from start to finish.
Memory Usage	KB	Peak memory consumption during execution, indicating efficiency of memory management.
CPU Cycles	–	Total number of CPU cycles consumed, providing a hardware-level measure of efficiency.
Throughput	–	Work completed per unit time; relevant for streaming or batch workloads.
Energy Consumption	Joules	Power usage recorded using Intel's RAPL interface, capturing energy efficiency.

TABLE I
QUANTITATIVE METRICS FOR EVALUATING SOURCE CODE OPTIMIZATION.

III. Problem Definition

A. Problem Statement

We study the problem of multi-objective software optimization. Given an input program C consisting of a function or class, the goal is to produce an optimized program C' that is functionally equivalent to C while improving its measured efficiency across a set of system-level objectives. These objectives include runtime, memory usage, CPU cycles, throughput, and energy consumption.

A valid solution must satisfy two requirements. First, it must preserve the semantics of the original program: for the same valid inputs, C' must produce the same outputs as C. Second, it should improve or maintain performance with respect to one or more target metrics, depending on the desired optimization preference. Since these objectives may conflict with one another, the optimization task is inherently multi-objective rather than a single-metric maximization problem.

Formally, let $m(C) = [m_1(C), m_2(C), \ldots, m_K(C)]$ denote the vector of measured performance metrics for program C, where each m_k corresponds to a metric such as runtime, memory usage, CPU cycles, throughput, or energy consumption. Given a user-specified or sampled preference vector $w = [w_1, w_2, \ldots, w_K]$, where $w_k \geq 0$ and $\sum_{k=1}^{K} w_k = 1$, the objective is to generate a functionally equivalent program C' that achieves a better weighted performance score than the original program under w.

B. Evaluation Metrics

To evaluate the effectiveness of our proposed model for source code optimization, we consider a comprehensive set of quantitative and qualitative metrics. These metrics capture not only the performance and efficiency of the optimized code but also its functional correctness and long-term maintainability.

1) Functional Correctness: The most fundamental requirement of code optimization is that the transformed code preserves the semantics of the original program. Any optimization that alters functionality is invalid. We measure correctness as a binary pass/fail outcome, where code is considered correct only if it both compiles and passes an established suite of test cases. This ensures that improvements in performance are not achieved at the cost of correctness.

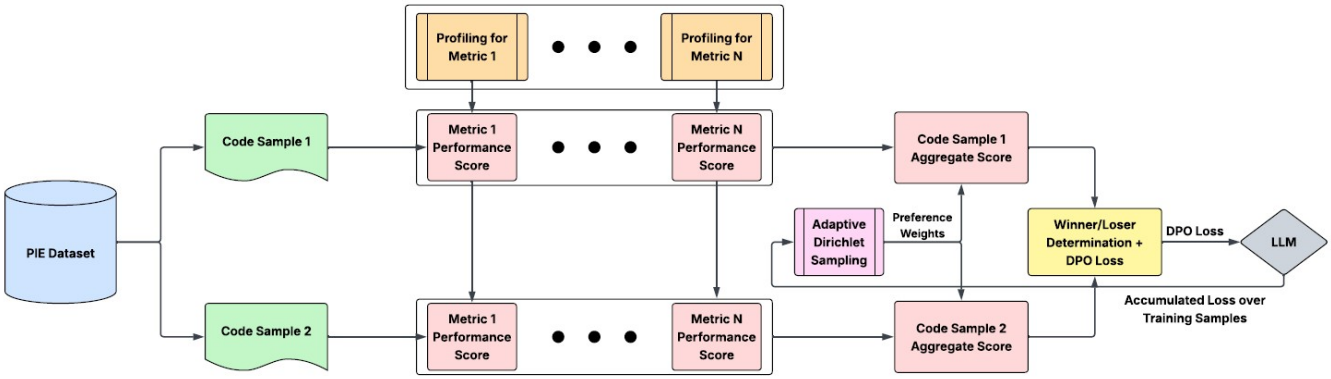

Fig. 1. Framework overview of Hydra

2) Runtime Performance Metrics: We employ several runtime metrics to quantify execution efficiency. Table I summarizes the metrics used.

3) Maintainability: While runtime performance is critical, overly aggressive optimizations often reduce code readability and interpretability. Transformations such as loop unrolling, tiling, or removal of syntactic sugar may hinder long-term maintainability. To account for this, we may evaluate code quality using large language models (LLMs) as automated reviewers. Optimized code is scored against existing industry coding standards to assess legibility, style adherence, and maintainability using metrics such as cyclomatic complexity. This ensures that the optimization process does not compromise future adaptability and developer productivity.

IV. Methodology

As shown above, Figure 1 illustrates the entire Hydra workflow, inspired by [24].

A. Stage 1: Dataset Construction

Hydra relies on a dataset containing pairs of functionally equivalent program submissions for programming tasks. Each pair consists of two implementations that produce correct outputs but differ in their implementation and performance characteristics. During training, we sample code-submission pairs for the same problem. The LLM does not generate code during training. Instead, the LLM is trained entirely offline using pre-existing code pairs. This design choice significantly reduces computational overhead and eliminates the need for code execution and correctness verification.

B. Stage 2: Preference Modeling via Dirichlet Distribution Sampling

To capture the diversity of optimization objectives typically encountered in real-world scenarios, we represent optimization preferences as a weight vector spanning multiple performance metrics. Let there be M metrics of interest, such as runtime, memory usage, CPU cycles, throughput, and energy consumption. We model user preferences as a probability simplex over these metrics and sample a preference vector

$$\mathbf{w} = (w_1, w_2, \ldots, w_M)$$

from a Dirichlet distribution parameterized by concentration parameters

$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_M).$$

The magnitude of an entry in α corresponds to the weight that particular metric receives in $\mathbf{w}$. The Dirichlet distribution ensures that the sampled weights sum to one, allowing them to be considered as relative importance values. The Dirichlet distribution is defined as

$$\mathbf{w} \sim \text{Dirichlet}(\boldsymbol{\alpha}), \qquad p(\mathbf{w} \mid \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{M} w_i^{\alpha_i - 1},$$

where $B(\boldsymbol{\alpha})$ is the multivariate Beta function. By varying $\boldsymbol{\alpha}$, the distribution can represent balanced preferences across all metrics, preferences that place high importance on a single metric, or intermediate preferences. Sampling from this distribution introduces stochasticity and diversity into the training data, thereby exposing the model to different optimization scenarios while avoiding overfitting to a single objective.

C. Stage 3: Profiling Code and Constructing Performance Vectors

Each code submission, C_i, in a given code pair is executed and profiled across the M metrics. Let $\mathbf{v}_i = (v_{i,1}, \ldots, v_{i,M})$ denote the raw performance vector. The vector is normalized using a Sigmoid function to map all metric values to be in the $[0, 1]$ range:

$$\tilde{\mathbf{v}}_i = \sigma(\mathbf{v}_i) = \frac{1}{1 + \exp(-\mathbf{v}_i)},$$

where the Sigmoid function $\sigma(\cdot)$ is applied element-wise.

The aggregate reward for code C_i given a sampled preference vector $\mathbf{w}$ from Stage 2 is then computed as the weighted sum of normalized metric values:

$$R_i = \mathbf{w} \cdot \tilde{\mathbf{v}}_i = \sum_{k=1}^{M} w_k \tilde{v}_{i,k}.$$

For each code pair (C_i, C_j), the submission with higher reward is labeled as the *winner*, C_w, and the other as the *loser*, C_ℓ. This information is provided to the next stage for calculating the DPO loss.

D. Stage 4: DPO Loss Calculation and LLM Fine-Tuning

We begin with an LLM and create two copies. The first is simply the original model, i.e., the reference model π_{ref}. The second is the policy model, which will be fine-tuned, π_θ. The DPO loss for a winner-loser pair (C_w, C_ℓ) can be written as:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\log \sigma\left(\beta\Big[\log \frac{\pi_\theta(C_w \mid x)}{\pi_{\text{ref}}(C_w \mid x)} - \log \frac{\pi_\theta(C_\ell \mid x)}{\pi_{\text{ref}}(C_\ell \mid x)}\Big]\right),$$

where:

- $\pi_\theta(C \mid x)$ is the probability of the code C given prompt x using the current LLM being fine-tuned,
- $\pi_{\text{ref}}(C \mid x)$ is the probability under the static reference model,
- β is a temperature/scaling parameter controlling the sharpness of the preference signal,
- $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.

The parameters of the LLM are then updated via Gradient Descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{\text{DPO}}(\theta),$$

where η is the learning rate.

E. Stage 5: Adaptive Dirichlet Distribution Sampling

We introduce a novel adaptive sampling module to guide the preference sampling towards metrics where the model is under-performing. We first classify a sampled preference vector $\mathbf{w}$ into a region $r \in \{1, \ldots, M, \text{balanced}\}$ based on the metric that dominates:

$$r = \begin{cases} k, & \text{if } w_k > \text{threshold} \\ M+1, & \text{otherwise (balanced region)} \end{cases}$$

For each region r, we compute a loss L_r, representing the model's performance given that preference vector. Over S samples in an epoch, the average loss per region is:

$$\bar{L}_r = \frac{1}{|\mathcal{S}_r|} \sum_{i \in \mathcal{S}_r} L_r^{(i)},$$

where $\mathcal{S}_r$ is the set of samples classified into region r.

Next, we calculate the mean loss across all regions:

$$\bar{L}_{\text{global}} = \frac{1}{R} \sum_{r=1}^{R} \bar{L}_r,$$

where R is the total number of regions.

Finally, the Dirichlet concentration parameters $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_R)$ are updated based on the deviation of each region's average loss from the global mean. This effectively performs Gradient Descent where the gradient is represented by the difference between the two loss terms:

$$\alpha_r \leftarrow \text{clip}\Big(\alpha_r \cdot \exp\left(\eta\left(\bar{L}_r - \bar{L}_{\text{global}}\right)\right),\, \alpha_{\min}, \alpha_{\max}\Big),$$

where η is a step size, and $\alpha_{\min}$ and $\alpha_{\max}$ are lower and upper bounds.

This update rule increases α_r for regions where the LLM performs below average, thereby increasing the probability of sampling preference vectors that emphasize under-performing metrics in the next epoch. Additionally, regions with below-average loss have their weights reduced, resulting in more focused training on metrics with weaker performance. After this update step, the following training iteration begins with the newly calculated Dirichlet concentration parameters.

F. Implementation:

For Stage 1, we utilized the PIE dataset for its format of optimized-unoptimized code pairs. This static dataset allows for offline training as specified in Stage 1. In our implementation of Stage 4, we started with the open-source model CodeQwen-1.5-7B-Chat using QLoRA, loaded in 4-bit NF4 with double quantization and kept frozen, as discussed earlier in this paper.

V. Experiments

A. Evaluating Hydra

We conduct experiments using code optimization pairs derived from the PIE dataset, which provides pairs of functionally equivalent implementations with differing performance characteristics. Each pair consists of a slower reference implementation and a faster alternative for the same task. In our experiments, we focus on Python code pairs to evaluate multi-objective inference-time optimization behavior across five system-level metrics: latency, memory usage, CPU cycles, throughput, and energy consumption.

Experiments use the enriched PIE Python split, where raw program pairs are scored across all five target metrics. The raw split contains 36,857 Python entries, and after enrichment and filtering, 35,752 usable preference pairs are used for DPO training. Each pair includes latency, peak memory usage, CPU cycles, throughput, and estimated energy. Energy is estimated using CPU cycles as cycles $\times\ 2.5 \times 10^{-10}$ J. This enriched dataset provides broader metric coverage than the earlier limited-metric setup, allowing the model to learn from multiple system-level optimization signals simultaneously.

Importantly, although all five metrics are present, they do not contribute equally to learning. Some metrics, such as latency, CPU cycles, throughput, and energy, provide stronger and more consistent preference signals, while memory usage is noisier and less reliable. This dataset property creates a challenging multi-objective setting where a well-designed framework must automatically identify and prioritize informative objectives without manual weight tuning.

To evaluate generalization, we construct held-out evaluation examples that are not used during training. This prevents memorization of specific code variants and instead evaluates

whether learned optimization behavior transfers to unseen programs with different inefficiency patterns.

B. Training

We fine-tune a pretrained code model, Qwen2.5-Coder-7B-Instruct, using Direct Preference Optimization (DPO) with LoRA adapters under a 4-bit NF4 QLoRA setup. The enhanced training run starts from a warm-start checkpoint trained on an earlier preference dataset and continues on the enriched Python preference pairs. The run is configured for up to 72,000 DPO steps using a learning rate of 2×10^{-6}, $\beta = 0.015$, and an adaptive Dirichlet learning rate of 0.002. The Dirichlet concentration parameters are clipped to $[\alpha_{\min}, \alpha_{\max}] = [0.75, 1.5]$.

Training is performed on a single NVIDIA A100 80 GB GPU using SLURM. Because long training jobs may be interrupted by scheduler time limits, the pipeline supports checkpoint resumption across multiple job slices. This allows training to continue from the latest saved state without discarding previous progress.

We compare two frameworks under identical settings. The Base Framework samples a Dirichlet weight vector once at dataset construction time and keeps it fixed throughout training. The Enhanced Framework samples weights per batch from a Dirichlet distribution parameterized by a learnable α vector, which is updated online using per-metric advantage signals. This baseline is intentionally chosen as a controlled ablation: both frameworks share the same model, data, and optimization settings, differing only in whether objective weights are static or adaptive. This isolates the contribution of adaptive Dirichlet sampling and α updates.

In the enhanced framework, preference labels are recomputed at each step using the current weights, allowing winner/loser assignments to evolve over training. This introduces an implicit curriculum where objectives with strong learning signals are emphasized, while weaker or noisier objectives are suppressed.

We track several internal signals to characterize learning dynamics. The DPO loss measures how well the policy assigns higher likelihood to preferred implementations relative to the reference model. The policy–reference divergence quantifies how far the learned policy deviates from the base model, serving as an indicator of stability and overfitting risk. Finally, we track per-metric advantage deltas Δ_k, which measure step-wise improvement for each objective. Unlike aggregate advantage, Δ_k reveals which metrics contribute to learning and whether the adaptive mechanism correctly shifts focus toward informative objectives over time.

C. Evaluation

We evaluate both frameworks at two levels: training-level behavior and inference-level performance on a held-out benchmark.

At the training level, we record DPO loss, policy–reference divergence, Dirichlet α trajectories, and per-metric advantage deltas throughout training. These signals are used to assess optimization stability, preference learning progress, and the behavior of the adaptive weighting mechanism.

At the inference level, we evaluate the trained model on seven held-out Python programs designed to represent common inefficiency patterns. These include closed-form algebraic simplification, nested-loop reduction, inefficient list construction, repeated string concatenation, manual dictionary counting, unnecessary file reads, and redundant sorting. This benchmark is designed to reflect realistic Python optimization scenarios rather than synthetic training conditions.

For each program, we compare the original and optimized implementations across all five system-level metrics: latency, peak memory usage, CPU cycles, throughput, and estimated energy consumption. Latency is measured using repeated wall-clock trials, memory is measured using peak allocation statistics, CPU cycles are collected using hardware performance counters, throughput is reported as function calls per second, and energy is estimated from CPU cycles. These metrics provide an external measure of optimization quality and allow us to assess whether improvements observed during training translate to meaningful performance gains at inference time.

Correctness is verified using deterministic test cases for each generated program. Since the small correctness benchmark is saturated by the evaluated models, correctness alone is not sufficient to distinguish optimization quality. Therefore, our evaluation focuses on whether functionally correct outputs also improve measured system-level performance.

This evaluation protocol enables us to compare the Base and Enhanced frameworks in terms of both learning dynamics and their ability to generalize optimization behavior to unseen Python programs, while isolating the impact of adaptive multi-objective preference weighting.

D. Empirical Study to Evaluate Baselines

1) Empirical Study Goals: In our efforts to develop a comprehensive study comparing our approach with other baselines, our first step was to conduct a smaller-scale empirical study. The primary goal of this study was to evaluate how leading code generation models respond to optimization objectives involving competing performance tradeoffs, specifically, runtime efficiency versus memory efficiency. Rather than evaluating only functional correctness, our objective was to measure whether different prompting objectives produce distinct optimization behaviors that form a meaningful runtime–memory tradeoff frontier.

Specifically, we investigated the following research questions:

- Can prompting alone steer LLMs toward different optimization objectives, such as runtime optimization, memory optimization, or balanced optimization?
- Do different granularities of the prompts (minimal versus detailed prompts) affect the optimization quality?
- How do open-source code optimization models compare against frontier proprietary models?

• Do existing state-of-the-art baselines naturally exhibit Pareto-style tradeoff behavior across runtime and memory objectives?

To answer these questions, we conducted a study using existing optimization baselines, including GPT-5.4 and Efficoder.

2) Baselines Tested: We used GPT-5.4 as a strong frontier-model baseline for code optimization tasks. The model was prompted directly to optimize code under different optimization objectives.

We also evaluated Efficoder, an optimization-focused open-source model. Following the methodology described in the original work, we fine-tuned a Qwen2.5-Coder-7B-Instruct backbone model using the author's precise training pipeline on their custom EffiInstruct code optimization dataset.

3) Benchmark Tasks: Experiments were conducted on the HumanEval code generation benchmark. For each task, models were asked to generate optimized implementations while preserving the original program behavior.

Each model was evaluated under three optimization objectives: Runtime-focused optimization, Memory-focused optimization, and Balanced optimization.

Additionally, two prompt granularities were evaluated: 1) Minimal prompts: short instructions specifying only the optimization objective. 2) Detailed prompts: longer prompts explicitly describing behavioral constraints and including optimization suggestions.

This resulted in six prompting configurations per model: 1) Runtime-Minimal, 2) Runtime-Detailed, 3) Memory-Minimal, 4) Memory-Detailed, 5) Balanced-Minimal, 6) Balanced-Detailed.

For each of these tasks, the runtime was measured using wall-clock execution time averages across repeated executions. Memory usage was measured using the GNU **time** utility, which reports maximum resident set size (RSS).

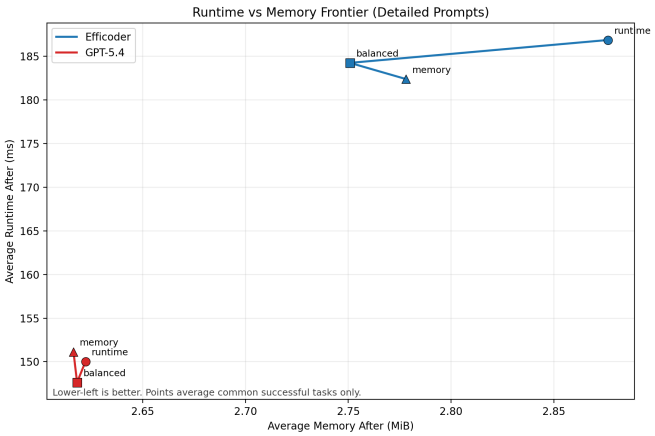

Fig. 2. Runtime vs Memory Pareto Frontiers for Detailed Prompts

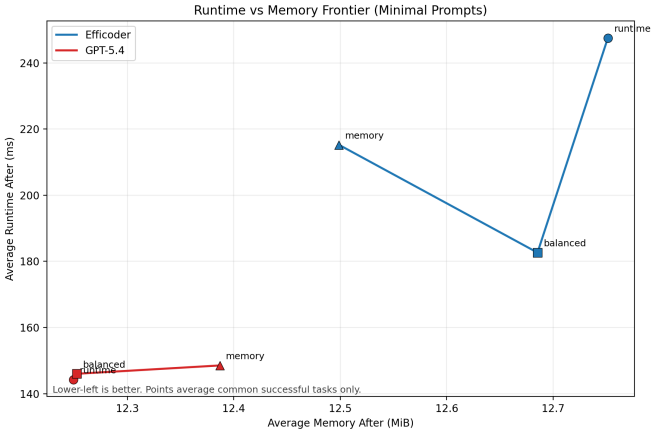

Fig. 3. Runtime vs Memory Pareto Frontiers for Minimal Prompts

VI. Results and Analysis

A. Analysis of the Baseline Methods' Performance

Figures 2 and 3 compare the runtime-memory tradeoffs from GPT-5.4 and Efficoder using both the detailed and minimal prompting strategies. Each point represents the average runtime and memory usage across tasks that successfully passed the functional correctness check. Lower-left positions indicate better overall performance.

Across both prompting strategies, GPT-5.4 consistently achieved substantially lower runtime and memory usage than Efficoder. The gap was especially evident for runtime, where GPT-5.4 reduced average execution time while also maintaining lower memory consumption. These results reinforced that frontier LLMs can perform strong optimization behaviors even without task-specific fine-tuning.

However, the objective-specific prompting did not produce a clean Pareto frontier. Prompts explicitly targeting memory optimization did not achieve the lowest memory usage, and runtime-focused prompts did not always minimize execution time. This behavior was observed for both GPT-5.4 and Efficoder. This indicates that current LLMs do not reliably separate optimization objectives through prompting alone, even when prompts are explicitly designed around runtime, memory, or balanced optimization.

Despite this, the balanced prompts frequently produced the strongest overall results, suggesting that models may implicitly optimize for general code quality rather than strictly adhering to a single optimization target. This behavior was particularly visible for GPT-5.4, where the runtime, memory, and balanced data points clustered closely together despite differing optimization instructions.

Additionally, the detailed prompting strategy generally shifted results toward lower memory usage values compared to the minimal prompting strategy, as seen by the leftward movement of points along the x-axis. This trend suggests that providing more explicit optimization guidance may help both frontier and smaller fine-tuned LLMs generate more efficient code.

Table II takes the results from the earlier Pareto analysis and presents them with more detail in tabular form. GPT-5.4 outperforms Efficoder on every metric in every condition, achieving substantially lower post-optimization runtimes and

TABLE II
EFFICODER VS. GPT-5.4 AVERAGE BEFORE/AFTER METRICS. COMPUTED ON COMMON SUCCESSFUL TASKS (N) OUT OF 163 TOTAL; BEFORE VALUES USE A SHARED AVERAGED BASELINE.

Objective	Model	Detail	N	Runtime Before (ms)	Runtime After (ms)	Runtime Ratio	Memory Before (MiB)	Memory After (MiB)	Memory Ratio
Runtime	Efficoder	minimal	74	203.338	215.600	0.950	9.864	10.257	0.892
	GPT-5.4	minimal	74	203.338	**147.616**	**1.412**	9.864	**9.865**	**1.000**
	Efficoder	detailed	78	224.098	202.473	1.062	9.094	9.264	0.942
	GPT-5.4	detailed	78	224.098	**150.931**	**1.516**	9.094	**9.105**	**0.998**
Memory	Efficoder	minimal	70	208.496	205.265	0.950	10.473	10.696	0.948
	GPT-5.4	minimal	70	208.496	**151.442**	**1.428**	10.473	**10.545**	**0.999**
	Efficoder	detailed	68	257.613	234.339	1.032	14.181	14.305	0.966
	GPT-5.4	detailed	68	257.613	**160.941**	**1.628**	14.181	**14.215**	**0.999**
Balanced	Efficoder	minimal	64	323.077	178.601	1.768	7.843	8.223	0.901
	GPT-5.4	minimal	64	323.077	**144.835**	**2.211**	7.843	**7.844**	**0.999**
	Efficoder	detailed	61	206.693	181.124	1.152	2.346	2.505	0.949
	GPT-5.4	detailed	61	206.693	**142.447**	**1.472**	2.346	**2.349**	**0.999**

near-identical memory footprints to the baseline.

The runtime improvements are most pronounced under the balanced objective, where GPT-5.4 achieves ratios of 2.211 and 1.472 for minimal and detailed prompting respectively, compared to Efficoder's 1.768 and 1.152. This suggests GPT-5.4 is better able to exploit joint runtime-memory tradeoffs when not constrained to a single objective.

On the memory objective, GPT-5.4 consistently achieves memory ratios near 1.000, meaning its optimized code consumes almost exactly the same memory as the raw code. Efficoder's memory ratios are trivially lower across the board (0.948–0.966). This pattern holds under both prompting conditions, and a manual inspection of the raw code revealed less opportunity for memory optimizations versus runtime.

Interestingly, the strong performance of both models under the balanced objective suggests that when optimization is driven purely by prompting, without any explicit training on tradeoffs, a balanced prompt may be preferable to a narrowly targeted one, since general-purpose LLMs are not inherently trained to reason about competing resource objectives. This motivates our proposed approach: by incorporating tradeoff-aware training via DPO, where preference pairs explicitly encode runtime-memory tradeoffs, we expect the model to develop a more principled understanding of the optimization landscape and consequently achieve better tradeoff navigation than prompt-engineering-based optimization alone can provide.

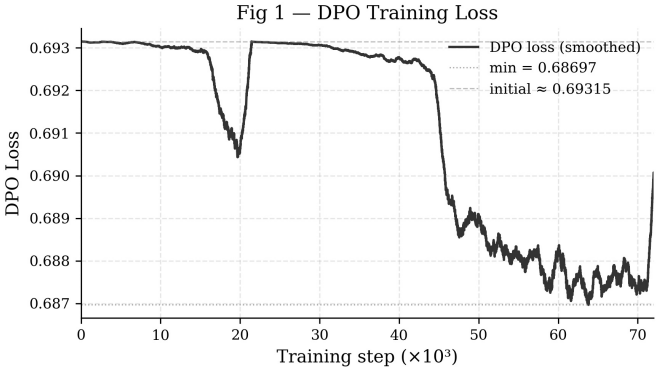

Fig. 4. DPO training loss during the enhanced training run. The stable loss trend indicates that adaptive multi-objective weighting does not destabilize preference optimization.

B. Analysis of Hydra with Adaptive Dirichlet Sampling

We evaluate the Hydra framework along two complementary dimensions: (i) training-time behavior under adaptive multi-objective weighting, and (ii) inference-time performance on unseen Python optimization tasks. This Python evaluation complements the broader experimental setup by testing whether the adaptive framework can produce functionally correct and performance-improving outputs under five measured or derived system-level metrics.

Figure 4 shows the DPO training loss during the enhanced training run. The loss remains stable throughout training, indicating that the adaptive Dirichlet mechanism can be integrated into DPO without causing collapse or unstable preference updates. This stability is important because the framework changes the relative contribution of each optimization objective during training instead of relying on fixed metric weights.

C. Adaptive Weighting Dynamics

Figure 5 shows the evolution of the Dirichlet concentration parameters throughout training. Unlike the static weighting baseline, the enhanced framework dynamically adjusts objective weights based on per-metric advantage signals observed during training.

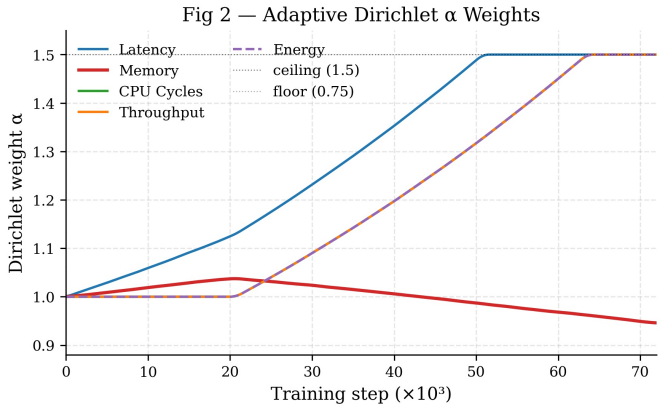

Fig. 5. Evolution of Dirichlet concentration parameters during training. Metrics with stronger and more consistent preference signals receive higher adaptive weight, while weaker or noisier objectives receive less emphasis.

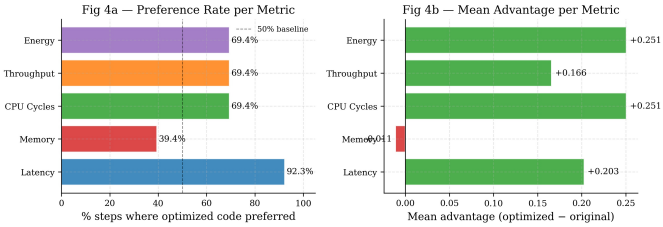

Fig. 6. Final per-metric preference behavior after training. Metrics with higher preference rates and stronger mean advantages contribute more strongly to the learned optimization policy.

The learned weighting behavior shows that the framework does not treat all objectives equally when the data does not support them equally. Metrics with consistent improvement signals, such as latency, CPU cycles, throughput, and energy, receive stronger weight during training. In contrast, memory receives lower relative emphasis when its advantage signal is weaker or noisier. This behavior allows the model to focus on objectives that provide reliable supervision rather than forcing the optimizer to learn equally from all metrics.

D. Final Objective Allocation

Figure 6 summarizes the final learned objective allocation. The resulting distribution is non-uniform, showing that the framework adapts to the quality and consistency of each metric signal. Latency receives the strongest preference signal, while CPU cycles, throughput, and energy also provide useful optimization feedback. Memory contributes less strongly, reflecting its weaker and more variable behavior in the training data.

These results indicate that the enhanced framework performs adaptive objective prioritization during DPO training. Instead of assuming that all optimization metrics are equally informative, the method learns which metrics provide reliable preference signals and adjusts the training objective accordingly.

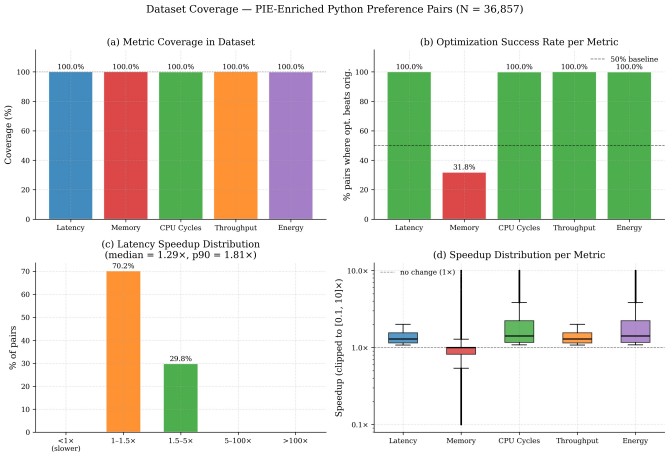

Fig. 7. Dataset coverage and optimization signal characteristics for the PIE-enriched Python preference dataset. (a) Coverage of the five optimization metrics across all enriched pairs. (b) Fraction of preference pairs for which the optimized implementation outperforms the original under each metric. (c) Distribution of latency speedups across preference pairs. (d) Speedup distributions for all five metrics, clipped to $[0.1\times, 10\times]$ for visualization.

E. Dataset Metric Coverage

Figure 7 provides a more detailed view of the enriched Python preference dataset. As shown in Fig. 7(a), all five target metrics latency, memory, CPU cycles, throughput, and energy are available across the enriched preference pairs, giving the adaptive framework access to a broad set of optimization signals. This is a substantial improvement over more limited settings in which only a subset of metrics is available.

However, broad coverage does not imply equally strong training signal. In Fig. 7(b), latency, CPU cycles, throughput, and energy show near-perfect optimization success rates, meaning that the optimized code in the preference pairs consistently outperforms the original under these metrics. Memory behaves differently, with a substantially lower success rate, indicating that memory improvements are weaker, less consistent, or more task-dependent in this dataset. This explains why the adaptive framework assigns memory less relative weight during training.

The latency distribution in Fig. 7(c) further shows that most preference pairs provide moderate but meaningful latency improvement, with the majority falling in the 1–1.5× range and a smaller fraction showing larger gains. Figure 7(d) extends this view to all five metrics. Latency, CPU cycles, throughput, and energy generally exhibit speedup distributions above the 1× baseline, whereas memory is centered closer to parity and shows higher variability. Taken together, these results show that the enriched dataset provides broad metric coverage, but that the quality of the optimization signal differs substantially across objectives. This motivates the use of adaptive weighting, which increases emphasis on reliable metrics and reduces the influence of weaker ones.

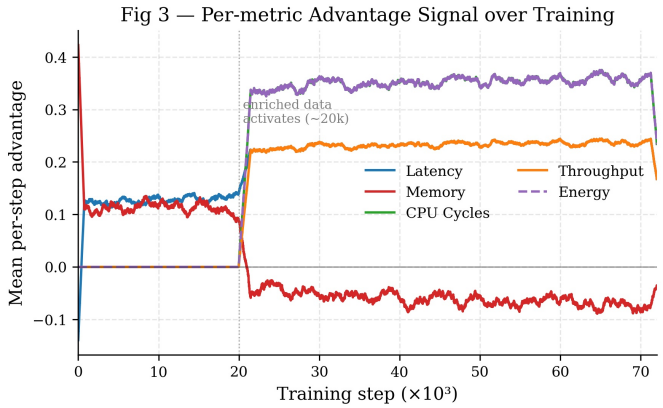

Fig. 8. Per-metric advantage signals during training. Stronger positive advantage values indicate objectives where the preferred code consistently outperforms the rejected code.

F. Distribution of Optimization Signals

Figure 8 illustrates the strength of the per-metric optimization signals used for adaptive weighting. Latency, CPU cycles, throughput, and energy show clearer positive advantage signals, indicating that the preferred programs are more consistently better under these objectives. Memory, however, provides a weaker signal, which explains why the adaptive framework assigns it less relative emphasis during training.

This result reinforces the need for adaptive weighting. In a multi-objective code optimization setting, different metrics do not contribute equally to preference learning. The proposed framework uses observed advantage signals to determine which objectives should receive stronger weight, reducing the need for manual weight tuning.

G. Inference-Time Performance and Generalization

To evaluate generalization, the trained model is tested on unseen Python programs exhibiting common inefficiency patterns. These include redundant loops, unnecessary sorting, repeated string concatenation, repeated file reads, manual dictionary counting, and inefficient list construction. The generated outputs are executed and compared against the original implementations using latency, peak memory usage, CPU cycles, throughput, and estimated energy consumption.

The results show that the model applies non-trivial optimizations in several cases. In particular, avoidable algorithmic overhead is removed in examples such as triangular summation and nested-loop filtering, resulting in large reductions in latency, CPU cycles, and estimated energy. Similarly, repeated file reads and redundant sorting are improved by replacing unnecessary repeated work with simpler computations.

However, the improvements are not uniform across all cases. Some transformations improve latency while increasing memory usage, as seen in the string concatenation case. Other examples show small regressions in wall-clock time while still improving CPU cycles and estimated energy. These cases highlight the importance of multi-metric evaluation: a model may improve one objective while slightly regressing another.

Overall, these findings provide preliminary evidence that the enhanced framework can generalize optimization behavior beyond the training examples while using adaptive objective prioritization. At the same time, the results reveal remaining challenges in balancing competing objectives and improving robustness across all code patterns.

H. Inference Results

Inference-time behavior was evaluated using a small set of Python programs designed to expose common optimization opportunities. Each program targets a specific inefficiency pattern, such as redundant computations, repeated loops, unnecessary file reads, repeated container operations, or inefficient string construction. Evaluation is performed by executing both the original and optimized programs and measuring latency, peak memory usage, CPU cycles, throughput, and estimated energy. Table III summarizes the observed results.

The results show that the enhanced framework can produce substantial performance improvements for several classes of inefficiencies. The largest gains occur when the original program contains avoidable algorithmic overhead. For example, the triangular-sum benchmark is transformed into a closed-form computation, producing near-zero measured latency and large reductions in CPU cycles and estimated energy. Similarly, the nested-loop example is simplified substantially, resulting in a major latency reduction and a large throughput increase.

Other examples show more moderate but still meaningful improvements. The file-line benchmark reduces both latency and memory usage by avoiding unnecessary repeated work. The redundant-sort and list-comprehension examples show smaller gains because the original implementations are already relatively efficient. The string concatenation case improves latency but increases memory usage, illustrating that optimizing one metric can sometimes regress another. This tradeoff is expected in multi-objective code optimization and motivates the use of adaptive metric weighting.

The Dict Histogram case shows a small latency regression while still reducing CPU cycles and estimated energy. Because the measured latency difference is small, this case is likely affected by runtime noise and Python interpreter overhead. This example highlights the importance of reporting multiple metrics rather than relying only on wall-clock time.

Overall, the inference results provide preliminary evidence that the DPO-trained policy, combined with adaptive Dirichlet weighting, can learn and apply meaningful optimization patterns beyond the training examples. The strongest gains occur when the target program contains clear algorithmic inefficiencies, while smaller or noisier cases remain more challenging. Since the evaluation set is small and all evaluated models pass the correctness suite, future work should evaluate on larger held-out benchmarks and include stronger correctness filtering and more precise performance measurement.

VII. Conclusion

In this work, we explored preference-based LLM fine-tuning for MOSO in an offline setting. Instead of relying on online

TABLE III

BEFORE/AFTER INFERENCE-TIME PERFORMANCE COMPARISON BETWEEN THE ORIGINAL PYTHON PROGRAMS AND THE PROGRAMS OPTIMIZED BY OUR MODEL. THE EVALUATED METRICS INCLUDE LATENCY, PEAK MEMORY USAGE, CPU CYCLES, THROUGHPUT, AND ESTIMATED ENERGY CONSUMPTION. ENERGY IS ESTIMATED AS CYCLES $\times\ 2.5\times10^{-10}$ J.

	Latency (ms)		**Memory (KB)**		**CPU Cycles**		**Throughput**		**Energy (mJ)**	
Sample	**Orig.**	**Opt.**	**Orig.**	**Opt.**	**Orig.**	**Opt.**	**Orig.**	**Opt.**	**Orig.**	**Opt.**
Triangular Sum	220.4	≈ 0	< 1	< 1	380M	26M	136 ops/s	39M ops/s	95.0	6.5
Nested Loop Filter	871.5	1.1	71	71	1.15G	29M	6 ops/s	5K ops/s	287.5	7.3
File Lines (2 reads)	11.1	3.2	1217	615	45M	36M	2K ops/s	6K ops/s	11.3	9.0
String Concatenation	25.3	15.0	107	1163	62M	56M	790 ops/s	1K ops/s	15.5	14.0
List Comprehension	276.9	245.2	19694	19694	338M	282M	72 ops/s	82 ops/s	84.5	70.5
Redundant Sort	79.7	75.1	10931	9368	99M	90M	251 ops/s	266 ops/s	24.8	22.5
Dict Histogram	144.1	147.8	6284	6303	360M	336M	139 ops/s	135 ops/s	90.0	84.0
Average / Summary	**46.7% latency reduction**		**9.1% memory reduction**		**36.0% cycles reduction**		**improved in most cases**		**36.0% energy reduction**	

Note: ≈ 0 indicates execution time below the timer resolution. String concatenation improves latency but increases memory usage because `join()` allocates a larger intermediate buffer. The Dict Histogram latency regression is small and likely within measurement noise.

code generation, execution, or correctness checking, we construct training preferences by profiling existing program pairs from the PIE dataset across multiple performance metrics. By sampling weight vectors from a Dirichlet distribution and applying them to normalized performance metrics, we generate a diverse set of implicit optimization objectives that get converted to binary preference pairs required for DPO. This approach avoids the substantial computational costs associated with online LLM/DPO fine-tuning techniques.

Our baseline evaluation reveals that strong general-purpose models such as GPT-5.4 already achieve meaningful runtime improvements through prompting alone, with runtime ratios as high as 2.211 under the balanced prompting strategy. However, both baseline approaches struggle to improve memory usage and neither demonstrates reliable tradeoff-aware behavior when more specific optimization objectives are specified. These results confirm that prompt-based optimization, while effective for single-objective tasks, is insufficient for principled multi-objective tradeoff navigation, reinforcing the need for explicit tradeoff-aware training.

Our experiments demonstrate some early promise. Exposing the model to a richer distribution of trade-offs and preferences during training can help it be more robust when performing MOSO during inference. This underscores the value of using a Curriculum Learning-like training approach.

VIII. LIMITATIONS

Our framework has several limitations that stem from its underlying design. First, Hydra relies on executing candidate implementations to obtain multi-objective performance measurements, resulting in nontrivial computational overhead that may limit scalability to large codebases or expensive workloads. Second, the quality of the learned optimization policy is limited by the diversity of the sampled candidates and objective weightings. An insufficient coverage can lead to biased preference signals and reduced generalization to unseen trade-offs. Additionally, while DPO enables stable offline preference learning, it lacks the ability to actively explore new optimization strategies beyond the provided dataset, which may restrict performance improvements in complex or highly dynamic environments. Also, our approach assumes that the profiling-based measurements are reliable and comparable across executions, while in practice, nondeterminism and environment-specific factors can introduce variance into the reward signal, affecting training stability and optimization quality.

Metric signal quality. Even when a metric is present in the dataset, it may provide a contradictory or near-random preference signal that does not reliably distinguish better from worse implementations. In our Python experiments, memory usage exhibited a preference rate of only 39.4% barely above chance meaning the model had no consistent basis for preferring memory-optimized code. While the adaptive Dirichlet mechanism correctly responds by down-weighting such metrics, this represents a data quality limitation rather than a framework limitation: collecting training pairs with stronger and more diverse memory optimization signals would be necessary to make memory a reliable training objective.

Benchmark saturation. The correctness benchmark used for evaluation is limited to a small set of seven handcrafted examples, all of which all evaluated models including the untuned base model solve correctly. This means correctness alone is insufficient to differentiate model quality at this scale; more discriminating evaluation on larger, harder, or real-world benchmarks (e.g., the full PIE test split) is needed to draw stronger conclusions about generalization.

Finally, for a new dimension to work within our framework it would ideally need to be: (1) a continuously measurable variable that can produce a scalar score per program, (2) consistently computable with a reliable automated tool, and (3) comparable across programs. These constraints are what allow our framework to succeed on the dimensions we selected. In contrast, a dimension such as security is largely boolean, and there is no established continuous scale on which to say one program is ”20% more secure” than another. This makes it very challenging to normalize against other dimensions or include it in a weighted tradeoff with other continuous variables.

IX. Future Work

There are several promising avenues to explore in the future, building on this work beyond conducting a more thorough analysis comparing Hydra to the baselines and an ablation study. First, future work could investigate explicit preference conditioning during inference. For example, learned preference embeddings or control tokens could be introduced into the prompt to enable users to steer the model toward particular optimization strategies without requiring retraining. Additionally, our use of categorical grouping and gradient descent is intentionally simple. Using more rigorous approaches could enable increased stability and optimal updates to the sampling process.

Finally, extending this framework beyond PIE to larger, more diverse codebases would further test its generalizability. Combining offline preference learning with limited online refinement may also serve as a promising hybrid approach. Future experiments should assess how optimizations hold up across varying input sizes and hardware environments.

X. Appendix

A. Related Works

Developing energy-efficient systems and software is a long-standing and well-established challenge in the computing domain. While this paper focuses primarily on reinforcement-learning-based approaches to software optimization, that is not the only method

ranging from traditional approaches that revolve around modeling computing systems and developing design methodologies and decision frameworks to optimize the system architecture [25]–[27] to AI approaches focused on Supervised Finetuning of LLM code optimization models.

1) Non-AI Approaches for Software Optimization: Traditional code optimization techniques primarily focus on transforming intermediary representations of source code at various stages of compilation, resulting in executables that are more efficient while preserving the original program semantics. High-level optimizations, such as constant propagation and constant folding [28], are applied early in the compilation process to reduce code redundancies and simplify computations. Mid-level optimizations, including loop unrolling [29], loop tiling [30], and common sub-expression elimination, aim to improve execution time by enhancing spatial and temporal locality, thereby reducing expensive memory access operations. At the low level, optimizations such as branch prediction hints [31] and tail call elimination operate on the low-level intermediate representation (LIR), providing final, platform-agnostic modifications that increase CPU/GPU throughput and reduce stack frame overhead. However, such heuristic optimization techniques may not adapt well to the dynamic workloads common in modern AI applications [32]. Because their instruction sets are often fixed and manually tuned, many techniques are limited in their ability to exploit complex runtime patterns as well as predict performance bottlenecks in highly parallelized or irregular computation graphs.

AI Approaches for Software Optimization:

*2) **Fine-Tuning Approaches**:* Recent advances in fine-tuning large language models (LLMs) for code optimization have introduced strategies that go beyond next-token prediction and traditional reward-based learning objectives. These approaches leverage architectural design choices, diverse training data, and latent representation learning to better align model outputs with performance-oriented goals such as execution speed and efficiency.

One such method is Problem-Oriented Optimization (POO), which frames optimization as a program synthesis task by fine-tuning on diverse human-generated solutions to the same problem [33]. This ensemble-style supervision captures a broad spectrum of optimization strategies, improving the model's generalization across problem domains. To ensure correctness, the method incorporates an Anchor Verification mechanism that compares generated code to slower but verified solutions. This helps the model maintain correctness while learning to generate more optimal solutions. However, the reliance on crowd-sourced code can introduce bias toward common strategies and may limit exploration of unconventional but efficient patterns.

Another approach involves encoder-decoder architectures, such as those used in the E-code model [34]. This method uses an Expert Encoder Group composed of multiple BERT-style encoders. These encoders separately process the problem description, an inefficient baseline implementation, and input/output examples. These embeddings are then fused and passed into a decoder (GPT-Neo) to generate optimized code. An execution time predictor filters the generated outputs during inference, selecting only those expected to run faster. Although effective in performance improvement, the model's complexity and reliance on execution prediction can increase computational overhead during inference.

Lastly, LLMs can also be fine-tuned using contrastive learning to align latent representations of optimized and unoptimized code. For example, while CodeT5+ also uses an encoder-decoder architecture, it further incorporates contrastive loss alongside standard generation objectives to pull semantically equivalent solutions closer in the embedding space [35]. This alignment of code representations enables the model to generalize efficiency across semantically equivalent and structurally diverse implementations, even when surface syntax varies. However, this approach lacks explicit performance feedback, which may limit its ability to capture low-level optimization effects.

*3) **Feedback-based Iterative Optimization Approaches**:* Traditional heuristic-based code optimization techniques are often limited in adaptability. Recent research instead introduces feedback-based iterative approaches that leverage LLMs to produce more dynamic and targeted optimizations. [36] [37] [38].

One such method is Preference Optimization with Runtime Feedback, which fine-tunes a model on self-generated preference data labeled as "quick vs. slow" and "passed vs. failed" to inherently produce code that is both correct and has a

lower execution time [36] [39]. The framework's effectiveness relies on starting with a small but high-quality set of problems and unit tests [36] [39]. Furthermore, all experiments were conducted using Python, and the results may not generalize to other programming languages without further analysis.

A second approach, Compiler-Informed Optimization, uses a compiler as an automated oracle to provide detailed feedback to an LLM, guiding it to reduce the instruction count of LLVM IR [36]. The feedback in this loop includes verification of predicted instruction counts and validation of optimization passes, which helps the LLM correct its internal understanding and improve code size. The feedback model struggles with generations produced at a higher temperature. It was trained on feedback from deterministic outputs (temperature 0), and its ability to provide useful corrections diminishes when the generated code is significantly different or more "fuzzy" [36].

A variant, the "Fast Feedback" loop, accelerates optimization by using a more concise feedback report. This avoids requiring the LLM to generate a full optimized IR and makes each iteration about 10× faster [36] While the "Fast Feedback" model can correct errors and improve upon a single, deterministic generation, it fails to improve the performance of the original model when 10 or more diverse samples are generated. The research found that simply sampling from the original model without any feedback loop achieves higher performance when allowed many attempts, indicating that for this specific problem, a well-tuned sampling strategy is more powerful than an iterative feedback loop [36].

Early Unsuccessful Iterations

B. Supervised Finetuning

For the goal for the first prototype, we aimed to assess whether supervised fine-tuning (SFT) can serve as the primary method for code optimization in our pipeline. We additionally sought to obtain a code-optimization dataset for the proposed pipeline and used this prototype to test it, with potential adoption for our pipeline contingent on promising results.

We assembled a dataset of instruction–response pairs drawn from the corpus of a recent paper called "HPC-Coder-v2: Studying Code LLMs Across Low-Resource Parallel Languages" by Chaturvedi, Nichols, Singh and Bhatele. Each prompt contained an unoptimized reference implementation and requested a functionally equivalent implementation with lower runtime. The target response supplied the corresponding optimized implementation used for supervision. This framing allowed us to test whether a model could learn to produce efficient code while preserving correctness.

For the model, we chose to fine-tune CodeQwen-1.5-7B-Chat using QLoRA. As of October 2025, this model was a top performing code generation model on HuggingFace via the Big Code Models Leaderboard. The base model was loaded in 4-bit NF4 with double quantization and kept frozen; LoRA adapters were the only trainable parameters, with bfloat16 compute. Each datapoint was rendered with the model's chat template, and we trained on all tokens in the conversation (system/user/assistant) under the standard causal-LM cross-entropy objective. Training used a small per-device batch with gradient accumulation, a 2048-token context, a cosine learning-rate schedule with warmup, the paged-AdamW (8-bit) optimizer, and gradient checkpointing to reduce memory. Checkpoints contained only the adapters, enabling compact single-GPU fine-tuning and inference.

For evaluation, we prioritized correctness and speed over text similarity. For each held-out task, we generated a candidate solution, normalized it into a callable entry point if needed, and ran unit tests against a task-specific oracle to verify functional equivalence. We then benchmarked the oracle, the model's candidate, and an unoptimized prompt baseline across increasing input sizes, reporting median runtimes. In a representative case (sum of 1..n), the fine-tuned model emitted the closed-form solution, passes all tests, and its runtime curve overlapped the oracle while decisively outperforming the loop baseline. These results indicated that the prototype does learn optimization patterns without sacrificing correctness, while retaining QLoRA's cost efficiency.

C. Reinforcement Learning - Proximal Policy Optimization

For the second prototype, our goal was to explore whether reinforcement learning (RL) can allow a model to learn code optimization directly from execution-based feedback. Instead of relying on labeled reference outputs, we aimed to fine-tune a pretrained code model through interaction with a reward signal that captures correctness, efficiency, and structural quality.

We used the `data/xlcost` benchmark as the training environment to study how Proximal Policy Optimization (PPO) adjusts a model's policy toward reward-driven behavior. In this setting, each generated program is executed and assigned a scalar reward derived from multiple criteria: whether the code compiles, passes unit tests, and yields an abstract syntax tree (AST) structurally similar to the reference implementation. This formulation encourages the model to generalize optimization patterns while maintaining functional equivalence.

The training framework followed the PPOCoding repository, which implements an RL loop through two key components: `rl_run.py`, managing rollouts and policy updates, and `reward.py`, defining the execution-based reward computation. The model begins by generating candidate completions for a given prompt; each completion is evaluated, and the reward is backpropagated to refine the model's generation policy under the PPO objective.

Initial setup encountered several technical challenges. Running the PPO loop locally led to dependency conflicts, including missing PyTorch, Tree-sitter, and SacreBLEU modules, as well as Tree-sitter API mismatches. Additional issues arose from parser build failures due to architecture incompatibility with Apple's M-series processors. After resolving these dependencies, the next step involves successfully executing the `run.sh` script to reproduce the full PPO training cycle and begin analyzing the resulting reward curves and CodeBLEU metrics. These outputs will help determine

whether reinforcement-based fine-tuning can effectively internalize reward signals and produce more efficient, correctness-preserving code compared to purely supervised methods.

D. Cross-Attention

Cross-Attention (CA) is a mechanism that allows one sequence of representations (Queries) to attend to another sequence (Keys and Values), enabling the modeling of relationships across different modalities. It is commonly used in tasks such as sequence-to-sequence translation. Motivated by this, we explored CA as a novel module to link program structure to the measured performance. Specifically, we applied CA between embeddings of individual lines of source code and a vector of normalized performance metrics (e.g., runtime, memory, energy), to encourage the model to learn which regions of code most strongly influence particular performance metrics.

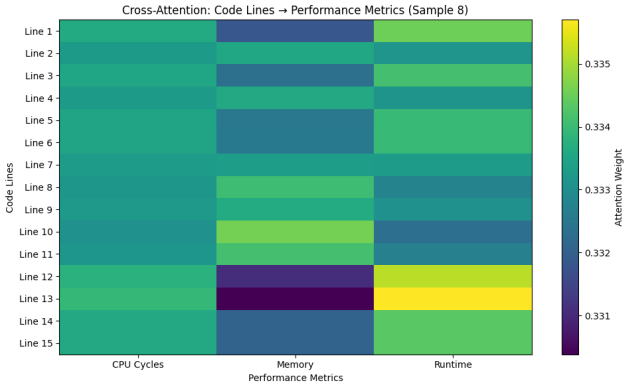

Fig. 9. Attention heatmap with uniformly distributed attention weights

However, our implementation yielded no meaningful results, and we soon realized that this approach is fundamentally limited in principle. Performance metrics are global properties of an entire program, while cross-attention assumes a localized relationship between Query and Key elements. We observed that the attention maps depicted a uniform distribution. When attending to the three performance metrics, each line of code assigns approximately equal weight (0.33) to all metrics, indicating that the attention mechanism fails to associate particularly strongly with any one of them. Figure 9 illustrates this effect, showing near-uniform attention across metrics for all code lines. This confirms that CA cannot learn meaningful associations in this particular use-case making it a poor tool choice for our problem.

E. Adaptive Dirichlet Sampling Proof-of-Concept Results

While we leave integrating the Adaptive Dirichlet Sampling module with the rest of the framework as future work, we developed a proof-of-concept implementation and demonstrated its functionality in an isolated simulation. The results are described below.

Figure 10 shows the evolution of the Dirichlet concentration parameters $\boldsymbol{\alpha}$ over 100 simulated epochs of adaptive sampling.

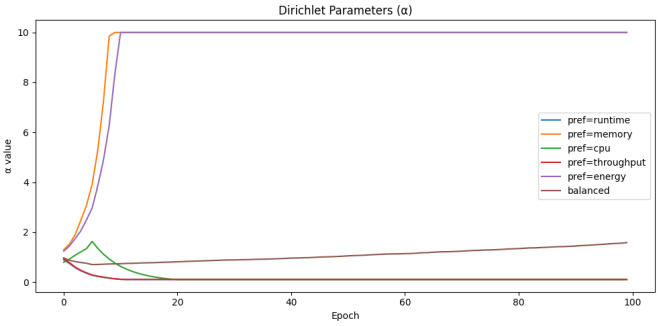

Fig. 10. Evolution of the Dirichlet concentration parameters over a simulated training period

Each α_k corresponds to one of the preference regions: runtime, memory, CPU, throughput, energy, and balanced, which means no particular preference. At the start of training, all parameters are initialized roughly equally to simulate a uniform prior over metric preferences. As training progresses, we observe that the α values associated with memory, CPU, and energy steadily increase, while those corresponding to runtime and throughput decrease or remain steady. In the simulation, we configured the DPO loss to be higher for the memory, CPU, and energy metrics and made it lower for runtime and throughput. Therefore, this trend demonstrates the effectiveness of the adaptive sampling strategy: regions where the simulated LLM performs poorly (i.e., higher DPO loss) are increasingly sampled, due to increasing α values, while regions where the LLM already excels are sampled less frequently. By epoch 50, memory and energy reach the hard-coded upper bound value of 10. The figure highlights the promise in the underlying theory of our Adaptive Dirichlet Sampling module as it shows the algorithm nudges the training to ensure that the LLM gets exposure to the most challenging metrics.

Figure 11 depicts the average DPO loss for each preference region throughout the course of training. Initially, the losses for memory, CPU, and energy regions are high, reflecting the intentionally poor performance of the simulated model on these metrics. Runtime and throughput start with lower losses as configured, indicating the model is already well-versed in those areas. During training, the adaptive sampling mechanism shifts the sampling distribution toward the high-loss regions, increasing their α values as shown in Figure 10. As a result, the average losses for memory and enegy begin to stabilize and decrease for CPU cycles. The balanced region maintains a moderate loss throughout, while runtime and throughput losses remain low. This behavior confirms that the adaptive mechanism is successfully focusing training on metrics that need improvement. Overall, these proof-of-concept results validate that the simulated DPO loss and adaptive sampling techniques can guide a model to prioritize learning about the more challenging metrics during training.

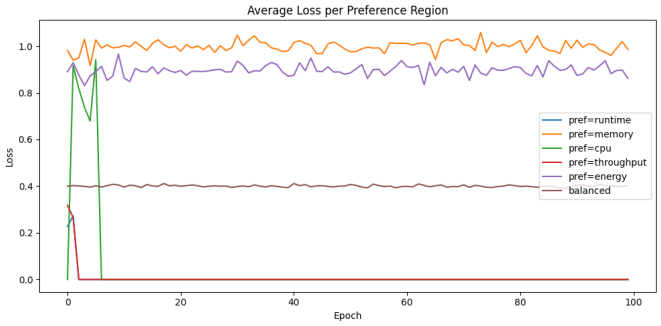

Fig. 11. Average loss per preference region over a simulated training period.

XI. Statement of Work

A. Arjun Gupte

During this semester, I focused on evaluating our framework against SOTA baselines. I surveyed recent literature, identifying and selecting baseline methods that were relevant and representative of our approach. I implemented the Efficoder and GPT-5.4 baselines and conducted all corresponding experiments (Section V.D) and data analysis (Section VI.A).

Beyond my technical work, I established the semester's big-picture goals and created concrete milestones to keep the team on track throughout. I organized and facilitated regular team meetings, led brainstorming sessions, and created slide decks presented during VIP classes covering complex AI and RL topics. I led authorship of multiple significant portions of the final research paper (Abstract, Introduction, Methodology, Experiments - Empirical Study, Results - Empirical Study, Conclusion, and Limitations). Throughout both semesters I provided ongoing mentorship by giving feedback on teammates' written work and presentations and sharing lessons from my prior research experience.

B. Ahmed Elmersawy

I was primarily responsible for implementing, stabilizing, and validating the Direct Preference Optimization (DPO) training and inference pipeline for our multi-objective code optimization framework. Early in the project, I implemented the preference scoring and reward aggregation components that convert measured code-pair performance into DPO-compatible winner/loser training data.

I then developed and refined the QLoRA-based DPO fine-tuning pipeline for training large code models efficiently. This included LoRA adapter integration, reference-policy handling, and stable DPO loss computation. A significant part of this work involved debugging SLURM configuration issues, CUDA memory limits, module mismatches, and environment inconsistencies to successfully fine-tune Qwen-based code models on large preference datasets.

For the Python model, I implemented the adaptive Dirichlet multi-objective weighting mechanism, which updates metric weights during training based on observed preference signals. I also built the enriched PIE data pipeline to score and filter preference pairs across five optimization metrics: latency, memory, CPU cycles, throughput, and energy. I managed long training runs, checkpoint resumption, and training stability across multi-day jobs.

I also conducted the model evaluation pipeline and designed and executed the five-metric inference benchmark on held-out Python programs using `perf stat` and `tracemalloc`,producing the results reported in Table III.

I also contributed extensively to writing the paper, including authoring the full experiments and results and analysis helped update the corresponding limitations.

C. Stefan Maxim

My contributions to the paper encompassed technical direction, empirical methodology, and manuscript preparation. I helped establish the project's research focus by identifying and systematizing a set of hardware-level code optimization techniques suitable for model training, with particular emphasis on transformations at the intermediate representation (IR) level across compilation stages. This work was supported by a review of relevant microarchitectural concepts, including branch prediction mechanisms and tail-call elimination, as well as established optimization strategies such as loop unrolling and loop tiling. I also investigated methods for quantifying performance and energy efficiency, incorporating tools such as Intel RAPL and Linux perf into our evaluation framework. In collaboration with the senior team lead and graduate researchers, I contributed to the decision to transition the project toward a fine-tuning-based approach, specifically leveraging preference-based methods rather than relying solely on inference-time optimization. In addition to shaping the technical direction, I was responsible for drafting several core sections of the manuscript, including the Abstract, Related Work, Metrics, and the section on Direct Preference Optimization (DPO). This required synthesizing existing literature on preference-based learning and adapting DPO to the context of code optimization. On the experimental side, I contributed to dataset construction by extending the PIE Madaan code-pair dataset with additional performance and energy-related annotations, and conducted evaluations across heterogeneous hardware environments, including personal computing systems, consumer GPUs, and Purdue's RCAC cluster. I also implemented the DPO-based training pipeline used in our experiments and produced quantitative visualizations to support our results. Finally, I authored a complementary paper documenting an agent-based implementation of the prior iteration of our optimization system; this work was accepted for publication in the Journal of Purdue Undergraduate Research and serves as a baseline for assessing the improvements introduced in the current study.

D. Andre Lee

As a direct contribution to Hydra's semester goal of documenting comparisons in performance with alternative baselines, I began the Spring 2026 semester by performing a literature review on the Efficoder, Afterburner, PIE, and Mercury research papers. Summarizing the definitions, results, methods, benefits, and limitations of each paper, I compiled my findings

into a shared online document. After discussing with the Hydra team, we decided to proceed with implementing the group relative policy optimization (GRPO) method discussed within the Afterburner paper as one of our selected comparison baselines.

We determined that our eventual goal is to implement a lightweight Docker-free alternative to Afterburner's methods that is able to utilize the CodeNet or PIE dataset, which we currently use with our DPO method. I worked on setting up the Monolith environment and replicating the results of the paper with these goals in mind. Near the end of this semester, I helped revise a portion of this paper in accordance with the editorial comments of Dr. Davis and reported on my findings on the Afterburner baseline in our Spring Undergraduate Research Conference slide presentation.

REFERENCES

[1] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, Apr. 1991.

[2] C. Wilke, S. Richly *et al.*, "Energy consumption and efficiency in mobile applications: A user feedback study," in *International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, 2013.

[3] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communications Surveys & Tutorials*, 2015.

[4] H. Krasner, "The cost of poor software quality in the US: A 2020 report," *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, vol. 2, p. 3, 2021.

[5] P. Garraghan, I. S. Moreno, Townend *et al.*, "An analysis of failure-related energy waste in a large-scale cloud environment," *IEEE Transactions on Emerging Topics in Computing*, 2014.

[6] V. Ferme and C. Pautasso, "Towards holistic continuous software performance assessment," in *ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. Association for Computing Machinery, 2017.

[7] Y. Liu and W. Meng, "Acquirer: A hybrid approach to detecting algorithmic complexity vulnerabilities," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. Association for Computing Machinery, 2022.

[8] The MITRE Corporation, "CWE-1132: Inefficient Algorithmic Complexity," https://cwe.mitre.org/data/definitions/1132.html, 2024, accessed: 2025-05-10.

[9] Z. Deng, Y. Guo, Han *et al.*, "AI agents under threat: A survey of key security challenges and future pathways," *ACM Comput. Surv.*, vol. 57, Feb. 2025.

[10] E. P. R. Institute, "Powering intelligence: Analyzing artificial intelligence and data center energy consumption," Electric Power Research Institute, Brochure Product ID 3002028905, 2024.

[11] European Commission, "Commission adopts EU-wide scheme for rating sustainability of data centres," 2024.

[12] S. Chen, "How much energy will ai really consume? the good, the bad and the unknown," *Nature*, 2025.

[13] C. Paulsen, J. Boyens, Bartol *et al.*, "Criticality analysis process model," National Institute of Standards and Technology, NIST Inter-agency/Internal Report (NISTIR) NISTIR 8179, 2018.

[14] A. Currie, S. Hsu, and S. Bergman, *Building Green Software*. O'Reilly Media, Inc., 2024.

[15] S. S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1998.

[16] D. A. Bader, B. M. E. Moret, and P. Sanders, *Algorithm Engineering for Parallel Computation*. Springer Berlin Heidelberg, 2002.

[17] R. Buchty, V. Heuveline, Karl *et al.*, "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators," *Concurrency and Computation: Practice and Experience*, 2012.

[18] D. Huang, G. Zeng, J. Dai, M. Luo, H. Weng, Y. Qing, H. Cui, Z. Guo, and J. M. Zhang, "Efficoder: Enhancing code generation in large language models through efficiency-aware fine-tuning," *arXiv preprint arXiv:2410.10209*, 2024.

[19] M. Du, L. A. Tuan, Y. Liu, Y. Qing, D. Huang, X. He, Q. Liu, Z. Ma, and S.-k. Ng, "Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization," *arXiv preprint arXiv:2505.23387*, 2025.

[20] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," 2022. [Online]. Available: https://arxiv.org/abs/2207.01780

[21] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy, "Execution-based code generation using deep reinforcement learning," 2023. [Online]. Available: https://arxiv.org/abs/2301.13816

[22] K. Zhang, G. Li, Y. Dong, J. Xu, J. Zhang, J. Su, Y. Liu, and Z. Jin, "Codedpo: Aligning code models with self generated and verified source code," 2025. [Online]. Available: https://arxiv.org/abs/2410.05605

[23] D. Nichols, P. Polasam, H. Menon, A. Marathe, T. Gamblin, and A. Bhatele, "Performance-aligned llms for generating fast code," 2024. [Online]. Available: https://arxiv.org/abs/2404.18864

[24] R. Gupta, R. Sullivan, Y. Li, S. Phatale, and A. Rastogi, "Robust multi-objective preference alignment with online dpo," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 26, 2025, pp. 27 321–27 329.

[25] S. te Brinke, S. Malakuti *et al.*, "A design method for modular energy-aware software," in *ACM Symposium on Applied Computing*, 2013.

[26] S. Te Brinke, S. Malakuti *et al.*, "A tool-supported approach for modular design of energy-aware software," in *ACM Symposium on Applied Computing*, 2014.

[27] I. Manotas, L. Pollock, and J. Clause, "Seeds: a software engineer's energy-optimization decision support framework," in *International Conference on Software Engineering*, 2014.

[28] J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, "Chapter 5 - source code transformations and optimizations," in *Embedded Computing for High Performance*, J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, Eds. Boston: Morgan Kaufmann, 2017, pp. 137–183. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128041895000053

[29] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, p. 345–420, Dec. 1994. [Online]. Available: https://doi.org/10.1145/197405.197406

[30] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread scheduling for cache locality," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, p. 60–71, Sep. 1996. [Online]. Available: https://doi.org/10.1145/248208.237151

[31] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 51–61. [Online]. Available: https://doi.org/10.1145/123465.123475

[32] Z. He, R. Abhyankar, V. Srivatsa, and Y. Zhang, "Cognify: Supercharging gen-ai workflows with hierarchical autotuning," 2025. [Online]. Available: https://arxiv.org/abs/2502.08056

[33] Y. Ye, Z. Chen, T. Lu, R. Zhang, P. Liu, and Y. Xu, "Problem-oriented optimization: Benchmarking and fine-tuning llms for code optimization," *arXiv preprint arXiv:2406.11935*, 2024.

[34] J. Pan, J. Jin, Z. Liang, Z. Lin, and Z. Yu, "E-code: Learning code optimization with expert encoders," *arXiv preprint arXiv:2408.12948*, 2024.

[35] Y. Wang, S. Chen, X. Ma, Y. Zhou, P. Yin, L. Tan, X. Chen, W. Chen, X. V. Kong, L. Zettlemoyer *et al.*, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[36] D. Grubišić, C. Cummins, V. Seeker, and H. Leather, "Compiler generated feedback for large language models," arXiv preprint arXiv:2403.14714, 2024, submitted on 18 March 2024. [Online]. Available: https://arxiv.org/abs/2403.14714

[37] M. Sepidband, H. Taherkhani, S. Wang, and H. Hemmati, "Enhancing llm-based code generation with complexity metrics: A feedback-driven approach," 2025. [Online]. Available: https://arxiv.org/pdf/2505.23953

[38] "Search-based software engineering: 16th international symposium, ssbse 2024, porto de galinhas, brazil, july 15, 2024, proceedings," in *Lecture Notes in Computer Science*, ser. LNCS, G. Jahangirova and F. Khomh, Eds. Cham, Switzerland: Springer Cham, 2024, vol. 14767.

[39] L. Gee, M. Gritta, G. Lampouras, and I. Iacobacci, "Code-optimise: Self-generated preference data for correctness and efficiency," 2025. [Online]. Available: https://arxiv.org/pdf/2406.12502