

LLM Program Optimization via Retrieval Augmented Search

Sagnik Anupam, Alexander Shypula, Osbert Bastani
University of Pennsylvania

Abstract

Recent work has demonstrated the potential of large language models (LLMs) for program optimization, a key challenge in programming languages. We propose a blackbox adaptation method called Retrieval Augmented Search (RAS) that performs beam search over candidate optimizations; at each step, it retrieves in-context examples from a given training dataset of slow-fast program pairs to guide the LLM. Critically, we find that performing contextual retrieval based on an LLM-generated natural language description significantly outperforms retrieval based on the source code. We also propose AEGIS, a method for improving interpretability by decomposing training examples into “atomic edits” that are significantly more incremental in nature. We show that RAS performs up to $2.06\times$ better than prior state-of-the-art blackbox adaptation strategies on optimizing C++ programs, and that AEGIS performs up to $1.37\times$ better while making significantly smaller edits. We also show that using RAS improves the mean runtime percentile of Python programs by 10.27 compared to baselines.

1 Introduction

Given the success of large language models (LLMs) in writing code, there has been significant interest applying them to programming tasks. A particularly interesting task is program optimization, a long-standing problem in programming languages. Recent work has shown that LLMs have difficulty with this task out-of-the-box (Shypula et al., 2024)—intuitively, data on program performance is simply not widely available in traditional training datasets, making adaptation necessary.

To address this problem, they propose the “Performance Improving Edits (PIE)” benchmark, and use it to test a number of carefully designed adaptation strategies to identify effective algorithms for improving performance, including *blackbox* (i.e. prompting-based) adaptation strategies such

as instruction prompting (Mishra et al., 2022), in-context learning (Brown et al., 2020), chain-of-thought prompting (Wei et al., 2022), and retrieval augmented generation (Lewis et al., 2020). They find *dynamic code retrieval* to be most effective; this approach retrieves a handful of slow-fast program pair examples from the training set at test time that are most relevant to the current instance (based on embedding similarity). These pairs are then used as in-context examples to prompt the LLM. Intuitively, this approach makes effective use of the training set, which contributes to its success.

This existing approach is “end-to-end” in the sense that it takes an input program and asks an LLM to directly output an optimized version of that program. However, it differs significantly from how modern compilers work. Rather than making edits inspired by a handful of end-to-end examples, they systematically modify the program through a series of *compiler passes*, each of which is designed to perform a specific kind of optimization. These optimizations are inspired by existing examples, but in a way that generalizes them so they apply to new programs. Thus, a natural question is whether breaking end-to-end optimization into more incremental steps can improve performance.

Inspired by modern compiler design, we propose a novel retrieval-based adaptation strategy, *retrieval augmented search* (RAS)¹, which combines two insights to improve dynamic retrieval. First, rather than retrieve based on the code itself, it uses *contextual retrieval*, where it retrieves examples from the training set based on an LLM-generated natural language description of the program, abstracting the core algorithms and data structures used by the program from how they are implemented on a superficial level. Second, rather than retrieve a fixed set of programs, we perform beam search by iter-

¹Code and data available at <https://sagnikanupam.com/papers/llmprogramoptimization>. Correspondence to first author: sanupam<at>seas.upenn.edu.

actively performing the retrieve-optimize-evaluate loop. These two techniques result in a state-of-the-art blackbox technique for adapting LLMs to program optimization.

However, this technique still produces large changes that can be hard to interpret. To further address this issue, we propose *Atomic Edit Guided Search* (AEGIS), an additional preprocessing step to distill generalizable insights from the training data. In particular, we prompt the LLM to decompose a single slow-fast program pair in the training set into a sequence of *atomic edits*, which are incremental modifications associated with a natural language description of the edit, and then explain why the edit might improve performance. The description is intended to be generalizable, abstracting away specifics of the training example from which they are derived. After generating a dataset of atomic edits and examples associated with each edit, when given a new program, we use RAS to first search over incremental edits to this program. Each edit to this program is achieved by retrieving the most relevant atomic edit in our database and then prompting the LLM to apply this atomic edit to the new program. We then perform beam search over sequences of incremental edits to select the resulting program that achieves the greatest performance gain while preserving correctness.

We evaluate our approach using the PIE benchmark (Shypula et al., 2024) for C++ program optimization and the Mercury benchmark (Du et al., 2024) for Python program optimization. On PIE, RAS significantly outperforms dynamic retrieval, a state-of-the-art blackbox adaptation strategy, achieving an $8.70\times$ average speedup compared to $4.23\times$ for dynamic retrieval using Qwen3-Coder. Furthermore, AEGIS achieves a $6.08\times$ average speedup using GPT-4o, while reducing the average edit size (measured by string edit distance) by 17% when compared to RAS (with GPT-4o) and by 30% when restricting to the first edit in the search process (which is the most substantial one). Hence, RAS performs upto $2.06\times$ better than dynamic retrieval, while AEGIS performs $1.37\times$ better. We also use RAS to achieve a state-of-the-art performance of $9.18\times$ average speedup using DeepSeek 3.2. Then, we show that by executing RAS on Mercury and comparing against our best-performing baselines, we can improve the mean runtime percentile by 10.27 for Qwen2.5-7B-Instruct, significantly narrowing its performance gap as compared to more recent reasoning models.

Related work. Code optimization has long been a problem of interest in programming languages. However, these approaches typically operate at a lower level of abstraction and are incapable of producing high-level optimizations such as changing algorithms and data structures. Thus, there has been recent interest in leveraging LLMs to augment existing, symbolic techniques. One approach that directly uses LLMs to perform program optimization is the Search-Based LLM (SBLLM) (Gao et al., 2024), which proposes an evolutionary search framework to iteratively optimize Python and C++ programs. However, in their framework, retrieval and search are not integrated, and they do not use contextual retrieval. Furthermore, they only report speedups of $1.55\times$ on the PIE benchmark (using GPT-4), so even the existing dynamic retrieval approach studied in PIE substantially outperforms their approach. Finally, Qiu et al. (2025) studies capabilities of LLMs for Python program optimization, finding significant gaps compared to human experts. We focus on optimizing C++ code since performance can be measured in a reproducible way using a simulator (Shypula et al., 2024).

Retrieval augmented generation is broadly known to improve code generation (Wang et al., 2024). The specific idea of dynamically retrieving relevant in-context examples from a larger training set was first proposed in Poesia et al. (2022) and was later shown to be highly effective for program optimization (Shypula et al., 2024). Recently, MapCoder (Islam et al., 2024) has shown that retrieving “previously seen” programming examples can improve code generation on the HumanEval benchmark. While contextual retrieval has recently been popularized for LLMs (Anthropic, 2024), the idea of annotating code to improve code search has long been studied extensively in software engineering. Older techniques such as Portfolio (McMillan et al., 2011) rely on information retrieval methods such as PageRank. The idea of automatically generating the natural descriptions for code snippets artificially was proposed in CoaCor (Yao et al., 2019), which trains an LSTM to generate natural language descriptions for use by a retriever.

2 Retrieval Augmented Search

We describe our retrieval augmented search (RAS) algorithm (summary in Figure 1 and Algorithm 1).

Problem formulation. In the program optimization problem, the goal is to take a program $p \in \mathcal{P}$ as

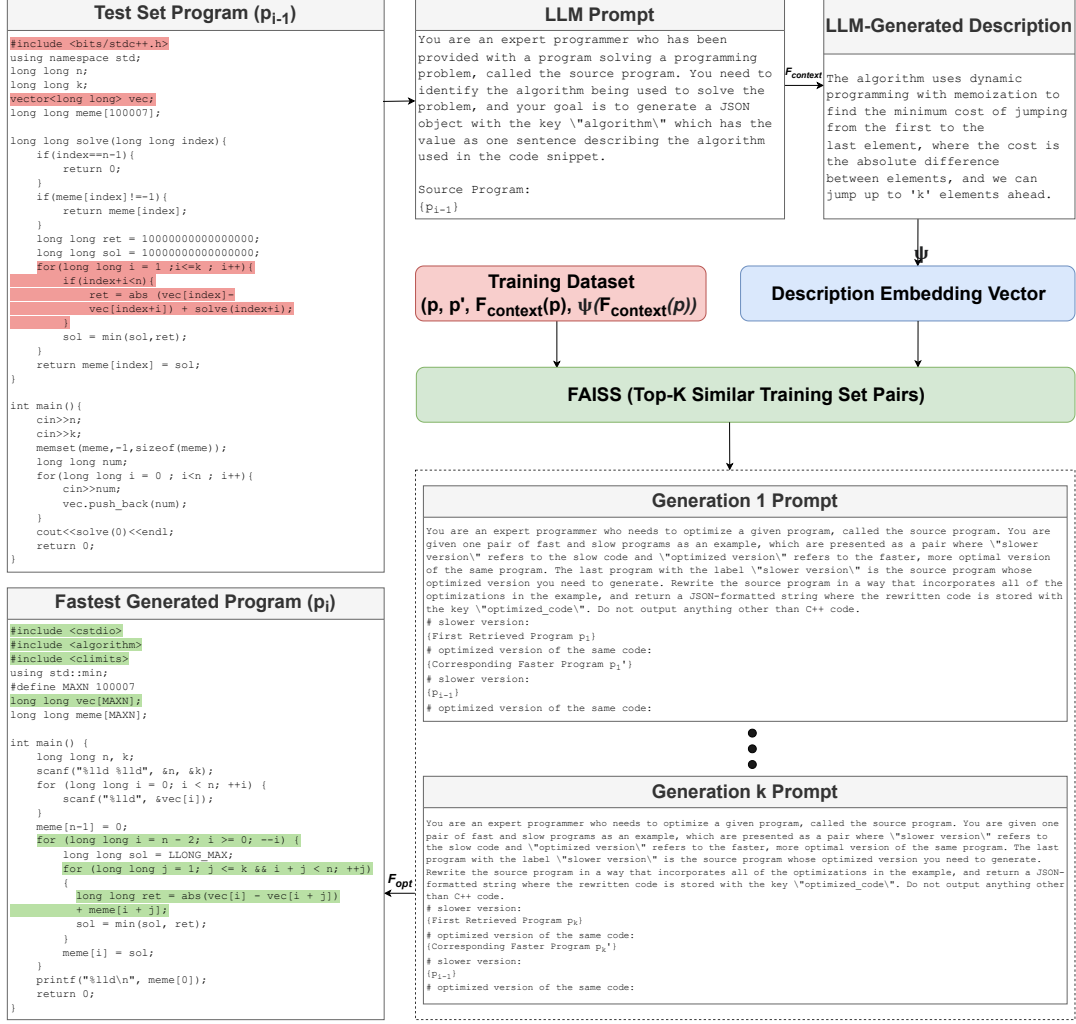


Figure 1: **RAS Framework**: For a given slow program p_{i-1} , we use $F_{context}$ to generate a program description and Ψ to generate its corresponding description embedding vector. We retrieve similar training set programs using FAISS and pass them to F_{opt} . The fastest program generated by F_{opt} is p_i .

Algorithm 1 Retrieval Augmented Search (RAS)

input: $p_0, \Pi_{train}, F_{opt}, F_{context}, R, \phi$
for $i \in [1, \dots, m]$ **do**
 $\Pi_i \leftarrow \text{top-}k\{((p, p'), d_\phi(p_{i-1}, p)) \mid (p, p') \in \Pi_{train}\}$
 $p_i^j \sim F_{opt}(\pi_i^j, p_{i-1}) \ (\forall j \in [k]) \quad \triangleright \Pi_i = \{\pi_i^j\}_{j=1}^k$
 $p_i \leftarrow \arg \max_{j \in [k]} R(p_i^j)$
return p_m

input, and output an optimized program $p' \in \mathcal{P}$ that is semantically equivalent to p . Typically, we are additionally given a set of test cases $\{(x_i, y_i)\}_{i=1}^k$ to check correctness; then, denoting the output of program p on input x as $p(x)$, we are searching for programs p such that $p(x_i) = y_i$ for all $i \in \{1, \dots, k\}$. While test cases do not guarantee semantic equivalence, they are widely used in machine learning for checking program equivalence

(Chen et al., 2021; Li et al., 2022).

For PIE, we focus on reducing running time, which we denote $R(p) \in \mathbb{R}$. Since we want the fastest correct program, we let $R(p) = -\infty$ if p does not pass one of the given test cases. In practice, measuring a speedup can be difficult due to the stochastic nature of program execution. Recent work has proposed benchmarks that seek to mitigate this issue. The approach used by the PIE benchmark is to measure performance using a system simulator (specifically, gem5 (Binkert et al., 2011)), which provides deterministic emulation of hardware, enabling fully reproducible results. Finally, we also set $R(p) = -\infty$ if evaluating p in gem5 times out. For Mercury, we set $R(p)$ to be a modified form of Beyond@1, their runtime percentile metric (described in Section 4.1).

To aid adaptation, we assume given a training set of slow-fast program pairs $\Pi = \{(p, p')\}_{j=1}^n$, where p is an unoptimized program and p' is a hand-optimized program; e.g., the PIE benchmark constructs such a dataset based on sequences of submissions from individual participants in competitive programming challenges (Shypula et al., 2024). Given a sequence of submissions p_1, \dots, p_k , they include pairs $(p_i, p_{i'})$ where $i < i'$ and where $p_{i'}$ is at least 10% faster than p_i according to gem5, i.e., $R(p_{i'}) \geq 1.1 \cdot R(p_i)$. They also provide a subset of *high-quality* training pairs that achieve a more substantial speedup by selecting a subset of the pairs $(p_{i'}, p_i)$ with the highest speedups $R(p_{i'})/R(p_i)$. Using their approach, we also construct a training set for Mercury by selecting high-quality pairs.

Finally, we are interested in blackbox adaptation techniques, which do not adjust the weights of the LLM; instead, they focus on prompting the LLM to improve performance. These prompts can be dynamic (e.g., include dynamically retrieved training examples), multi-turn (e.g., iteratively refine an example based on feedback), or incorporate search (e.g., incrementally apply a sequence of prompts).

General framework. We describe the general Retrieval-Augmented Search (RAS) framework for program optimization. RAS assumes that it is given a training set $\Pi_{\text{train}} = \{(p, p')\}_{j=1}^n$ of slow-fast program pairs, and a new program $p_0 \in \mathcal{P}$ to be optimized. In addition, it assumes it is given a retrieval strategy, which can be expressed as a distance function $d : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$ between pairs of programs. Typically, the strategy is defined by an embedding model $\phi : \mathcal{P} \rightarrow \mathbb{R}^d$, in which case we can define the distance based on the L_2 distance between the embedding vectors of two programs:

$$d_\phi(p, q) = \|\phi(p) - \phi(q)\|$$

Our framework also assumes blackbox access to an LLM F_{opt} , which takes as input an in-context example of a slow-fast program pair $\pi \in \mathcal{P}^2$, along with a new program p . Then, we can sample optimized versions $p' \sim F_{\text{opt}}(\pi, p)$ of p from F_{opt} . In our implementation, F_{opt} is provided with a system prompt instructing it to try and optimize p .

Now, RAS performs a variation of beam search to optimize p_0 , where at each step, it additionally retrieves in-context examples from the training set Π_{train} . In particular, at the i th iteration of beam search (starting from $i = 1$), let p_{i-1} be the current program. Then, we retrieve the top k programs

from Π_{train} to form the in-context dataset:

$$\Pi_i = \text{top-}k\{((p, p'), d(p_{i-1}, p)) \mid (p, p') \in \Pi_{\text{train}}\}.$$

Here, top- k selects the k *new* slow-fast pairs (p, p') with the smallest distances $d(p_{i-1}, p)$, using FAISS (Douze et al., 2024) for vector search. For any retrieved example π_i^j , we call π_i^j a new pair if F_{opt} did not use π_i^j to sample an earlier best-performing program $p_{\text{opt}} \in \{p_1, \dots, p_{i-1}\}$. Note that retrieval is performed based on the slow program p ; intuitively, we want a slow program that is similar to p_{i-1} so we can apply similar optimizations to p_{i-1} as the ones encoded by the pair (p, p') . Now, for each retrieved example $\pi_i^j \in \Pi_i$, we sample an optimized version $p_i^j \sim F_{\text{opt}}(\pi_i^j, p_{i-1})$ of p_{i-1} using π_i^j . Finally, we choose p_i to be the fastest program that correctly passes all test cases:

$$p_i = \arg \max_{j \in [k]} R(p_i^j),$$

where $[k] = \{1, \dots, k\}$. If no program passes all of the test cases (i.e., $R(p_i^j) = -\infty$ for all $j \in [k]$), or if all programs time out, then we set $p_i = p_{i-1}$. We continue this process for m steps, producing a sequence of programs p_1, \dots, p_m . Finally, we return p_m . If there is no program at step m that passes all of the test cases and does not time out, we return the source program p_0 . The hyperparameters of our algorithm are the number of in-context examples k and the number of iterations m ; we describe the choices we use in our experiments in Section 4.

Contextual retrieval. Our instantiation of RAS uses contextual retrieval to identify relevant in-context examples. We compute $\phi(p)$ by first using an LLM F_{context} to generate a natural language description (i.e., the “context” in contextual retrieval) of p (denoted $s = F_{\text{context}}(p)$), and then applying an embedding model ψ to obtain a vector $\psi(s) \in \mathbb{R}^d$, i.e., $\phi(p) = \psi(F_{\text{context}}(p))$. For examples $(p, p') \in \Pi_{\text{train}}$, we can precompute the embeddings, so the LLM F_{context} only needs to be run once for each one. To construct F_{context} , we use a blackbox LLM that is instructed to describe features like the algorithms and data structures used by the program; this prompt is shown in Figure 1 with an example of a pair (p, s) of program p and its description s . Finally, we also compare to an ablation used in prior work (Shypula et al., 2024) where we directly embed the given program—i.e., $\phi(p) = \psi(p)$ for some embedding model ψ ; we call this approach *code retrieval*.

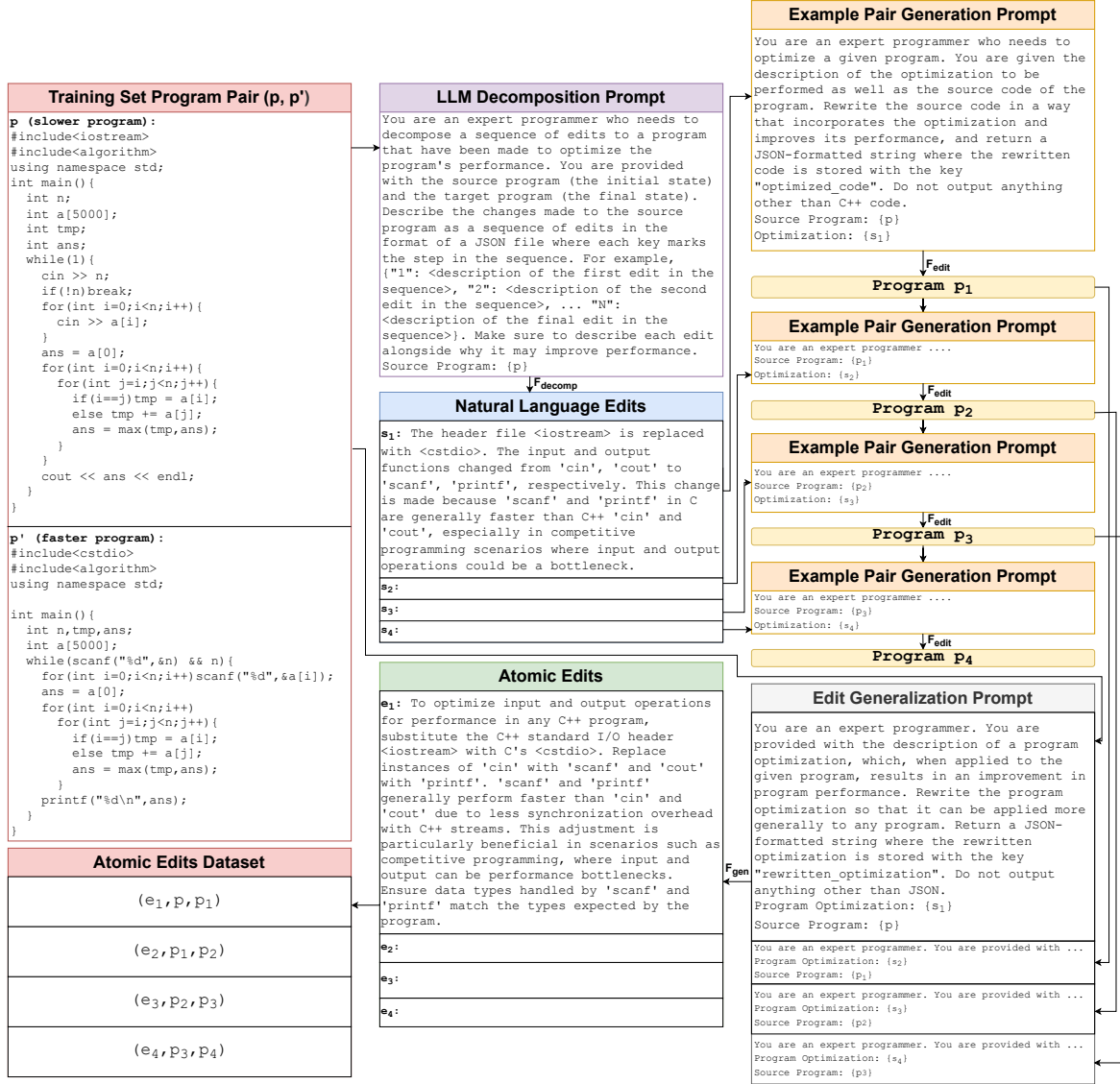


Figure 2: **AEGIS Framework:** For a given training set program pair (p, p') , we identify the natural language edits using F_{decomp} , and then generate intermediate programs implementing each edit by using F_{edit} . Finally, the natural language edits are generalized by F_{gen} to construct atomic edits.

3 Atomic Edit Guided Search

Next, we describe *Atomic Edit Guided Search* (AEGIS), which is a dataset preprocessing step designed to improve interpretability of RAS (overview in Figure 2 and pseudocode in Algorithm 2). AEGIS is inspired by modern compilers, which are designed to perform a sequence of *passes*, which incrementally transform the program to improve performance, which can improve interpretability since the changes from one step to the next may be easier to understand. We propose to generate *atomic edits*, which are pairs of programs (p, p') that are semantically equivalent and roughly differ by a single code optimization.

To this end, AEGIS replaces the original training

set Π_{train} with a set of atomic edits Π_{atomic} , and then uses RAS with Π_{atomic} . By retrieving atomic edits, we can guide the underlying LLM F_{opt} to perform incremental optimizations rather than large changes. AEGIS constructs Π_{atomic} by using an LLM F_{decomp} to decompose each pair $(p, p') \in \Pi_{train}$ into atomic edits, and then aggregates all discovered atomic edits to form Π_{atomic} .

Specifically, we instruct F_{decomp} to describe the differences between the each slow-fast pair $(p, p') \in \Pi_{train}$ as a list; then, the output of F_{decomp} is a list of *natural language edits* $[s_1, \dots, s_r] \sim F_{decomp}(p, p')$, where each s_i is a natural language description of an edit in (p, p') . Next, we apply each edit in sequence to the slow program p to

Algorithm 2 Atomic Edit-Guided Search (AEGIS)

input: $\Pi_{\text{train}}, F_{\text{decomp}}, F_{\text{edit}}, F_{\text{gen}}, F_{\text{opt}}, F_{\text{context}}, R$
 $\Pi_{\text{atomic}} \leftarrow \emptyset$
for $(p, p') \in \Pi_{\text{train}}$ **do**
 $[s_1, \dots, s_r] \sim F_{\text{decomp}}(p, p')$
 for $i \in [1, \dots, n]$ **do**
 $p_i \sim F_{\text{edit}}(s_i, p_{i-1})$
 $e_i \sim F_{\text{gen}}(s_i, p_i)$
 $\Pi_{\text{atomic}} \leftarrow \Pi_{\text{atomic}} \cup \{(e_i, (p_{i-1}, p_i))\}$
return Π_{atomic}

obtain a sequence of programs. We do so by initializing $p_0 = p$, and then prompting an LLM F_{edit} to apply natural language edit s_i to p_{i-1} to obtain the next program $p_i \sim F_{\text{edit}}(p_{i-1}, s_i)$ in the sequence; here, F_{edit} is instructed to apply the edit to the given program. Assuming the natural language edits accurately describe how p' is obtained from p , then the final program p_r in this sequence should resemble the original optimization p' of p ; i.e., p_r should also be an optimized version of p .

We construct our atomic edit dataset using pairs from the resulting sequence. For each tuple (s_i, p_{i-1}, p_i) , we ask an LLM F_{gen} to generalize s_i so it applies to a wider variety of programs; the resulting description $e_i \sim F_{\text{gen}}(s_i, p_i)$ is an atomic edit. Then, our dataset of atomic edits is

$$\Pi_{\text{atomic}} = \bigcup_{(p, p') \in \Pi_{\text{train}}} \{(e_i, (p_{i-1}, p_i))\}.$$

Finally, we can use RAS with Π_{atomic} , with a slight modification to account for some of the extra information. Specifically, we modify the LLM F_{opt} for program optimization to include the atomic edit—i.e., given atomic edit (e, π) and program p , we sample an optimized version $p' \sim F_{\text{opt}}(e, \pi, p)$. Intuitively, e provides instructions on how to optimize p , and π shows one example applying e .

4 Experiments

4.1 Experimental Setup

Benchmark. Our experiments are based on the PIE benchmark (Shypula et al., 2024), a dataset of slow-fast C++ program pairs constructed from submissions by human programmers to CodeNet (Puri et al., 2021); they execute C++ code in the gem5 simulator (Binkert et al., 2011) to measure running time. We use a similar method as PIE to construct training and test sets from Mercury (Du et al., 2024), a dataset of LeetCode problems with Python solutions; see Appendix A for details.

Baselines. We compare our approach to dynamic retrieval, the highest performing blackbox

adaptation strategy studied in PIE (Shypula et al., 2024). This approach also dynamically retrieves in-context examples from Π_{train} . There are two key differences between our approach and theirs. First, they use retrieval based on the embedding of the code itself rather than contextual retrieval (i.e., code retrieval). Second, they do not perform sequential search; instead, given a program p , they retrieve k in-context examples $\Pi \subseteq \Pi_{\text{train}}$ to provide to the LLM F'_{opt} , and then take multiple samples $p^1, \dots, p^h \sim F'_{\text{opt}}(\Pi, p)$. They return the fastest correct program among the h choices.

In addition, we also compare to a “no contextual” ablation of our approach that uses PIE’s strategy for retrieval but with search; in particular, it performs code retrieval instead of contextual retrieval. One iteration proceeds as with dynamic retrieval, but we perform multiple iterations. In particular, let p_0 be the initial program; on the i th iteration (starting from $i = 1$), we sample k in-context examples $\Pi \subseteq \Pi_{\text{train}}$ using code retrieval, draw samples $p_i^1, \dots, p_i^h \sim F'_{\text{opt}}(\Pi, p_{i-1})$, and then let $p_i = \arg \max_{j \in [h]} R(p_i^j)$; as in RAS, we let $p_i^j = p_{i-1}^j$ if $R(p_i^j) = -\infty$ for all $j \in [h]$.

We also consider a “Instruct Only” approach from PIE that performs neither retrieval (i.e., does not use Π_{train}) nor search; instead, we instruct the LLM F''_{opt} to optimize the given program p to obtain an optimized version $p' = F''_{\text{opt}}(p)$, i.e., F''_{opt} is an unadapted LLM. The prompt used in the “instruct only” setting is described in Appendix C, and the remaining prompts are described in Appendix D. Finally, we include the “human” speedup—for an initial program p , it is the speedup achieved by the fastest correct program p' written by the human participant who wrote p . For Mercury, we evaluate on our strongest-baseline, No Contextual, and provide our Instruct Only results for reference.

Hyperparameters. For all our PIE experiments, we incur a fixed evaluation cost since we evaluate 32 programs per approach. In our approaches (RAS and AEGIS with contextual retrieval), we use $k = 8$ retrievals and $m = 4$ beam search steps and take $h = 1$ sample per generated prompt. For our baselines, we normalize computation according to the number of calls to the LLM $F_{\text{opt}}, F'_{\text{opt}},$ or F''_{opt} . In this calculation, note that for F'_{opt} , the number of retrievals $k = |\Pi|$ does not affect the number of calls $F'_{\text{opt}}(\Pi, p)$, since all examples are included in a single call. Then, for our dynamic retrieval baseline, we retrieve $k = 4$ examples (the same as

Approach	GPT-4o		Qwen-3-Coder		DeepSeek 3.2	
	Mean Best Speedup	% Optimized	Mean Best Speedup	% Optimized	Mean Best Speedup	% Optimized
RAS	8.03	0.9692	8.70	0.9856	9.18	0.9908
No Contextual	5.84	0.8613	4.93	0.7667	7.62	0.9548
Dynamic Retrieval	4.43	0.8191	4.23	0.7749	7.03	0.9270
Instruct Only	2.31	0.5447	1.73	0.4018	2.89	0.5940
Human	3.63	0.9887	3.63	0.9887	3.63	0.9887

Table 1: Comparing RAS to baselines on PIE.

Approach	GPT-4o		Qwen-3-Coder		DeepSeek 3.2	
	Mean Best Speedup	% Optimized	Mean Best Speedup	% Optimized	Mean Best Speedup	% Optimized
AEGIS	6.08	0.9065	5.37	0.8469	6.48	0.9507
No Contextual	3.86	0.7585	2.76	0.4224	5.87	0.8695
Instruct Only	2.31	0.5447	1.73	0.4018	2.89	0.5940
Human	3.63	0.9887	3.63	0.9887	3.63	0.9887

Table 2: Comparing AEGIS to baselines on PIE.

used in PIE) and take $h = 32$ samples. For our “no contextual” ablation, we retrieve $k = 4$ examples, take $h = 8$ samples per iteration, and use $m = 4$ iterations (the same as our approach). For our “instruct only” ablation, we take $h = 32$ samples and use $m = 1$ iterations. We use k to denote the number of retrieved examples used in the prompt, as in (Shypula et al., 2024). For Mercury, we only execute $m = 2$ iterations of RAS (with $k = 8$, $h = 1$) and no contextual (with $k = 4$, $h = 8$), so we set $h = 16$ for the Instruct Only approach.

Metrics. Running gem5 on all test cases to evaluate a single program can be prohibitively computationally expensive (for compute specifications, see Appendix B).. Instead, we measure running time averaged across a subset of 5 randomly selected test cases; these 5 test cases are fixed ahead-of-time. To validate this strategy, we check the correlation between running times on the full test suite vs. our 5 random test cases across all programs in the PIE test set; we find a strong correlation (Pearson’s $r = 0.89$, $p < 0.001$; Spearman’s $\rho = 0.86$, $p < 0.001$), suggesting that 5 test cases suffices to obtain an accurate estimate of running time. We report results on the held-out test set $\Pi_{\text{test}} \subseteq \mathcal{P}$ of 973 unoptimized programs provided by the PIE benchmark. Our main metric is “mean best speedup”

$$\text{Speedup}(p, p') = \max \left\{ \frac{\text{RunningTime}(p')}{\text{RunningTime}(p)}, 1 \right\}$$

of the final program p' compared to the original pro-

gram p , averaged across all test programs $p \in \Pi_{\text{test}}$, where the minimum speedup is set to 1 since we can always return p . We also report “% Optimized”, which is the number of test programs p for which the optimized program p' is at least $1.1 \times$ as fast as p . While this metric is not the main goal of our system, it helps capture the diversity of programs that can be optimized using a given approach. For Mercury, we report their Pass@1 and Beyond@1 (mean runtime percentile) metrics (Du et al., 2024), with the modification that fastest generated program using each approach is assumed to be the 1 sample generated by the LLM.

4.2 Results

We show C++ results for RAS in Table 1 and for AEGIS in Table 2, and show Python optimization results for RAS in Table 3. First, note that RAS significantly improves performance compared to all baselines, when using both the original PIE training set as well as our atomic edit training set. Dynamic retrieval was by far the best blackbox adaptation approach studied in the original PIE paper, yet our approach is able to double its performance in terms of mean best speedup. Our ablation demonstrates that both search and contextual retrieval are roughly equally important, since ablating contextual retrieval about halves the performance improvement compared to dynamic retrieval. While AEGIS diminishes performance, it usually achieves a significant improvement. Indeed, for GPT-4o and Qwen-3-Coder, it outperforms all ablations (both

Method	Beyond@1	Pass@1
RAS	87.85	98.83
No Contextual	69.26	97.27
Instruct Only ($h=16$)	77.58	98.44
Base	58.66	96.88

Table 3: RAS Experiments on Mercury. The base values represent the unoptimized test set programs, while the others represent Qwen2.5-7B-Instruct experiments.

ablations of AEGIS and those of RAS); the only approach it does not outperform is the full RAS approach. We hypothesize that the performance gap between AEGIS and RAS occurs because 1) LLM-decomposed AEGIS program pairs are not guaranteed to have large differences in speedup and 2) multiple program pairs derived from the same RAS dataset pair may be selected during AEGIS’s retrieval, affecting retrieved program diversity. Using our PIE experiments, we study the impact of multiple rounds of search on our metrics in Appendix E.1, and the types of programming problems that RAS and AEGIS fail to optimize in Appendix E.3. We also study the impact of using specialized code embedding models instead of text-embedding-3-large in Appendix E.5. For Python optimization, we observe that RAS increases the mean runtime percentile metric (Beyond@1) by 10.27 for Qwen2.5-7B-Instruct, significantly narrowing the gap between it and the Instruct Only performance of larger models. Pass@1 and Beyond@1 results for larger models in the Instruct Only setting are provided in Appendix E.6.

Accuracy. RAS and AEGIS are designed to generate programs that pass all test cases; however, this strategy does not ensure correctness. To quantify the error rate of RAS and AEGIS more rigorously on PIE in our GPT-4o experiments, we examined if the programs selected at each step of the procedure would differ if at each iteration of search, we selected the fastest program while ignoring correctness. Across four iterations of search, while selecting for fastest program without measuring accuracy, for RAS, 5/973 test set instances choose an incorrect program (so accuracy is 99.5%), while for AEGIS, 0/973 test set instances chose an incorrect program (so accuracy is 100%). These results suggest that the LLM is highly accurate at producing optimizations that preserve semantic equivalence. We note that LLM-based program optimization systems are already deployed in practice (Shypula et al., 2025), leaving it up to the programmer to validate correctness of the optimizations.

To further assess accuracy, we used the C Bounded Model Checker (CBMC) (Kroening and Tautschnig, 2014) to formally analyze whether the optimized program is equivalent to the original one. We selected a subset of 10 programs generated by DeepSeek 3.2 using RAS; this limitation is due to the substantial manual effort required to format programs in a way that is amenable to CBMC. Specifically, we (1) filtered out programs that use C++ libraries not supported by CBMC, (2) selected program pairs in random order, and (3) rewrote the program pairs to use only functions supported by CBMC and to record program output in string buffers, without changing their semantics. Then, we selected the first 10 programs that do not cause out-of-memory issues when running CBMC. We observe that for 8 of the 10 programs, the model checker asserts that the generated program is equivalent to the original one; upon manual inspection, the 2 failures are due to minor differences, where one program introduced extra newline statements compared to the original, with the core semantics being equivalent. These results suggest that RAS-optimized programs are highly accurate.

Interpretability. A key motivation for AEGIS is that it should provide greater interpretability by making smaller edits. We consider two metrics. Our main metric is the character-level edit distance of pairs of programs (p_i, p_{i+1}) encountered as part of the search process, with lower edit distances indicating more incremental changes; we consider the edit distance averaged across all pairs of programs and across all programs in the test set. We summarize results for AEGIS and RAS in Table 5 in Appendix E.4, including results for the “no contextual” ablations of each approach. As can be seen, AEGIS significantly reduces mean edit distance in both cases. Furthermore, in Figure 3 in Appendix E.1, we show how the mean edit distance changes across steps. AEGIS significantly reduces mean edit distance in the first step, from about 500-600 to 250-400. These results suggest that RAS is performing significant optimizations in the first step, and the subsequent steps have smaller edit distance since the optimizations are more incremental.

5 Conclusion

We have proposed RAS and AEGIS, two methods for LLM-guided program optimization that incorporate beam search and retrieval to iteratively optimize a given program. We achieve significant

speedups in the blackbox setting (i.e., without any fine-tuning), outperforming existing LLM-based program optimization techniques. We believe that our approach provides a compelling strategy for adapting LLMs to code optimization.

Limitations. A key limitation of both our approaches is that they are more computationally expensive to execute due to our use of beam search. AEGIS also requires additional training-time compute since it uses LLM-generated code to construct its atomic dataset. Additionally, we surmise that it will be challenging to scale up our results to more complex, object-oriented codebases comprising several individual components since that would likely involve intermediate steps that identify the specific code snippets to be optimized. Nevertheless, we believe our methods pave a promising path towards effective application of LLMs to code optimization in practice.

Acknowledgements

This work is funded in part by the UPenn DARPA MOCHA CR PO-0074571/10101353 subcontract of Peraton Labs. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or Peraton Labs.

References

- Anthropic. 2024. Contextual retrieval. <https://web.archive.org/web/20250121234912/https://www.anthropic.com/news/contextual-retrieval>. Accessed: 2025-01-23.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, and 1 others. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281*.
- Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37:16601–16622.
- Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 254–266. IEEE Computer Society.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. **MapCoder: Multi-agent code generation for competitive problem solving**. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.
- Daniel Kroening and Michael Tautschnig. 2014. Cbmc–c bounded model checker: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120.
- Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. 2022. Reframing instructional prompts to gptk’s language. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 589–612.
- Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchronesh: Reliable code generation from pre-trained language models. In *The Tenth International Conference on Learning Representations*.
- Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian

- Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.
- Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2025. How efficient is llm-generated code? a rigorous & high-standard benchmark. *The Thirteenth International Conference on Learning Representations*.
- Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2025. Automated high-level code optimization for warehouse performance. *IEEE Micro*.
- Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. [Learning performance-improving code edits](#). In *The Twelfth International Conference on Learning Representations*.
- Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.
- Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The world wide web conference*, pages 2203–2214.

A Dataset Construction

For PIE (Shypula et al., 2024), we use 4080 high-quality pairs provided in the original dataset as our training set Π_{train} , and 973 test set pairs as a held-out test set Π_{test} . These high-quality pairs are constructed by taking up to 4 pairs in the PIE benchmark’s training set with the highest speedup for each competitive programming problem. Importantly, the train-test split in PIE is based on the competitive programming problem being solved, so the training and test set programs are semantically different. PIE has been used under a Apache License 2.0.

For Mercury, we are provided with a training and test set with reference solutions for each problem. We use the same approach on the 1633 LeetCode problems in the training set to construct a high-quality training set Π_{train} of 6372 pairs using LeetCode’s reported runtimes for the solutions. We evaluate our approaches on the slowest-provided reference solutions for the 256 held-out problems in Mercury’s test set (Du et al., 2024). Mercury has been used under a Creative Commons Attribution Non Commercial 4.0 license.

B Compute

For all experiments, we use OpenAI’s gpt-4o-2024-08-06 as F_{context} for the training set, as well as F_{decomp} , F_{edit} , and F_{gen} for the PIE atomic edit dataset. We then use the model specified in the experiment as F_{opt} , F'_{opt} , F''_{opt} , and F_{context} while executing the search procedure. The models used are Qwen-3-Coder (480B parameters) and DeepSeek V3.2 (685B parameters) We use OpenAI’s text-embedding-3-large as the embedding model ψ . We run the gem5 simulator on a server with $2 \times$ Intel(R) Xeon(R) Gold 6342 CPUs (96 cores total). All C++ programs evaluated in our experiments are compiled using a g++ compiler with the -O3 flag. We use an AWS t2.2xlarge instance for measuring runtime percentiles for Mercury.

C Comparing Instruction Prompting and Expert Programmer System Roles

In our “Instruct Only” baseline, we experiment with two prompts: an instruction-prompting approach (as described in the results of the original PIE benchmark (Shypula et al., 2024)), and an “expert programmer” system role. We provide the exact prompts for our approaches here and whenever

we refer to programs or retrieved natural language optimizations, we enclose them in braces. Our prompts are as follows:

C.1 Instruction Prompting (IP)

Given the program below, improve its performance:

```
### Program: {Program to be optimized}
```

```
### Optimized Version:
```

C.2 Expert Programmer System Role (EPSR)

System Role: You are an expert programmer who needs to optimize a given program. You are given the source code of the program. Rewrite the source code in a way that optimizes performance such that the program executes faster, and return a JSON-formatted string where the rewritten code is stored with the key “optimized_code”. Do not output anything other than C++ code.

User Role: Source Code: {Program to be optimized}

C.3 Prompt Result Comparison

We evaluate the two prompts on our dataset of 973 programs by taking $k = 32$ samples for $m = 1$ iteration of search. Our results are presented in Table 4.

Approach	Mean Best Speedup	% Optimized
EPSR	2.31	0.5447
IP	2.16	0.5632

Table 4: Results comparing differences in metrics due to prompts in Instruct Only setting

Since we observe a slight increase in Mean Best Speedup in the setting with an expert-level system role, we use it in all our other prompts for to maximize efficacy. The “Instruct Only” setting results we report in Tables 1 & 2 use this expert-programmer system role prompt, which is used by F''_{opt} .

D Prompts for Experimental Results

We present our prompts for our PIE experiments below. For our Mercury experiments, we replace all instances of the phrase “C++” with “Python”.

D.1 RAS

D.1.1 Program Description Generation

This prompt is used by F_{context} .

System Role: You are an expert programmer who has been provided with a program solving a programming problem, called the source program. You need to identify the algorithm being used to solve the problem, and your goal is to generate a JSON object with the key "algorithm" which has the value as one sentence describing the algorithm used in the code snippet.

User Role: Source Program:
{Program to be optimized}

D.1.2 Generating Programs With Contextual Retrieval

This prompt is used by F_{opt} .

System Role: You are an expert programmer who needs to optimize a given program, called the source program. You are given one pair of fast and slow programs as an example, which are presented as a pair where "slower version" refers to the slow code and "optimized version" refers to the faster, more optimal version of the same program. The last program with the label "slower version" is the source program whose optimized version you need to generate. Rewrite the source program in a way that incorporates all of the optimizations in the example, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.

User Role:

slower version:
{Retrieved Slow Program}
optimized version of the same code:
{Retrieved Faster Program}

slower version:
{Program to be optimized}
optimized version of the same code: \n

D.1.3 Generating Programs With Dynamic Code Retrieval

This is the prompt used in the "No Contextual" and "Dynamic Retrieval" settings for RAS, and the "No Contextual" setting for AEGIS. It is passed to F'_{opt} .

System Role: You are an expert programmer who needs to optimize a given program, called the source program. You are given several pairs of fast and slow programs, called examples, which are presented as pairs where "slower version" refers to the slow code and "optimized version" refers to the faster, more optimal version of the same program. The very last program with the label "slower version" is the source program whose optimized version you need to generate. Rewrite the source program in a way that incorporates all of the optimizations in the examples, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.

User Role:

slower version:
{Retrieved Slow Program 1}
optimized version of the same code:
{Retrieved Faster Program 1}

slower version:
{Retrieved Slow Program 2}
optimized version of the same code:
{Retrieved Faster Program 2}

slower version:
{Retrieved Slow Program 3}
optimized version of the same code:
{Retrieved Faster Program 3}

slower version:
{Retrieved Slow Program 4}
optimized version of the same code:
{Retrieved Faster Program 4}

slower version:
{Program to be optimized}
optimized version of the same code: \n

D.2 AEGIS

D.2.1 Generating Natural Language Edits

This prompt is used by F_{decomp} .

System Role: You are an expert programmer who needs to decompose a sequence of edits to a program that have been made to optimize the program’s performance. You are provided with the source program (the initial state) and the target program (the final state). Describe the changes made to the source program as a sequence of edits in the format of a JSON file where each key marks the step in the sequence. For example, “1”: <description of the first edit in the sequence>, “2”: <description of the second edit in the sequence>, ... “N”: <description of the final edit in the sequence>. Make sure to describe each edit alongside why it may improve performance.

User Role:

Source Program: {Slow Program from Training Set Program Pair}

Target Program: {Faster Program from Training Set Program Pair}

D.2.2 Generating Program Sequence from Natural Language Edits

This prompt is used by F_{edit} .

System Role: You are an expert programmer who needs to optimize a given program. You are given the description of the optimization to be performed as well as the source code of the program. Rewrite the source code in a way that incorporates the optimization and improves its performance, and return a JSON-formatted string where the rewritten code is stored with the key “optimized_code”. Do not output anything other than C++ code.

User Role:

Source Program: {Previous Program in Sequence}

Optimization: {Optimization to be applied to generate next program in the sequence}

D.2.3 Generating Atomic Edits from Natural Language Edits

This prompt is used by F_{gen} .

System Role: You are an expert programmer. You are provided with the description of a program optimization, which, when applied to the given program, results in an improvement in program performance. Rewrite the program optimization so that it can be applied more generally to any program. Return a JSON-formatted string where the rewritten optimization is stored with the key “rewritten_optimization”. Do not output anything other than JSON.

User Role:

Program Optimization: {Natural Language Edit}

Program: {Program in program sequence that the edit was applied to}

D.2.4 Generating Programs With Contextual Retrieval

This prompt is used by the modified F_{opt} when generating programs with AEGIS.

System Role: You are an expert programmer who needs to optimize a given program, called the source program. You are given the description of an optimization that is to be performed on the given program, as well as an example showing how to apply the optimization on an example program (called the example source) to get a target program (called the example target). Rewrite the source code in a way that incorporates all of the optimizations, and return a JSON-formatted string where the rewritten code is stored with the key “optimized_code”. Do not output anything other than C++ code.

User Role: Source Program:

{Program to be optimized}

Optimization:

{Atomic edit retrieved via contextual retrieval}

Example Source:

{Slower program in retrieved example pair}

Example Target:

{Faster program in retrieved example pair}

E Additional Experimental Results

E.1 Metrics Across Beam Search Iterations

In Figure 3, we study the effect of using search techniques by reporting our various metrics across iterations of beam search on our C++ experiments. We focus on our results for our approach compared to our “No Contextual” ablation (since “Dy-

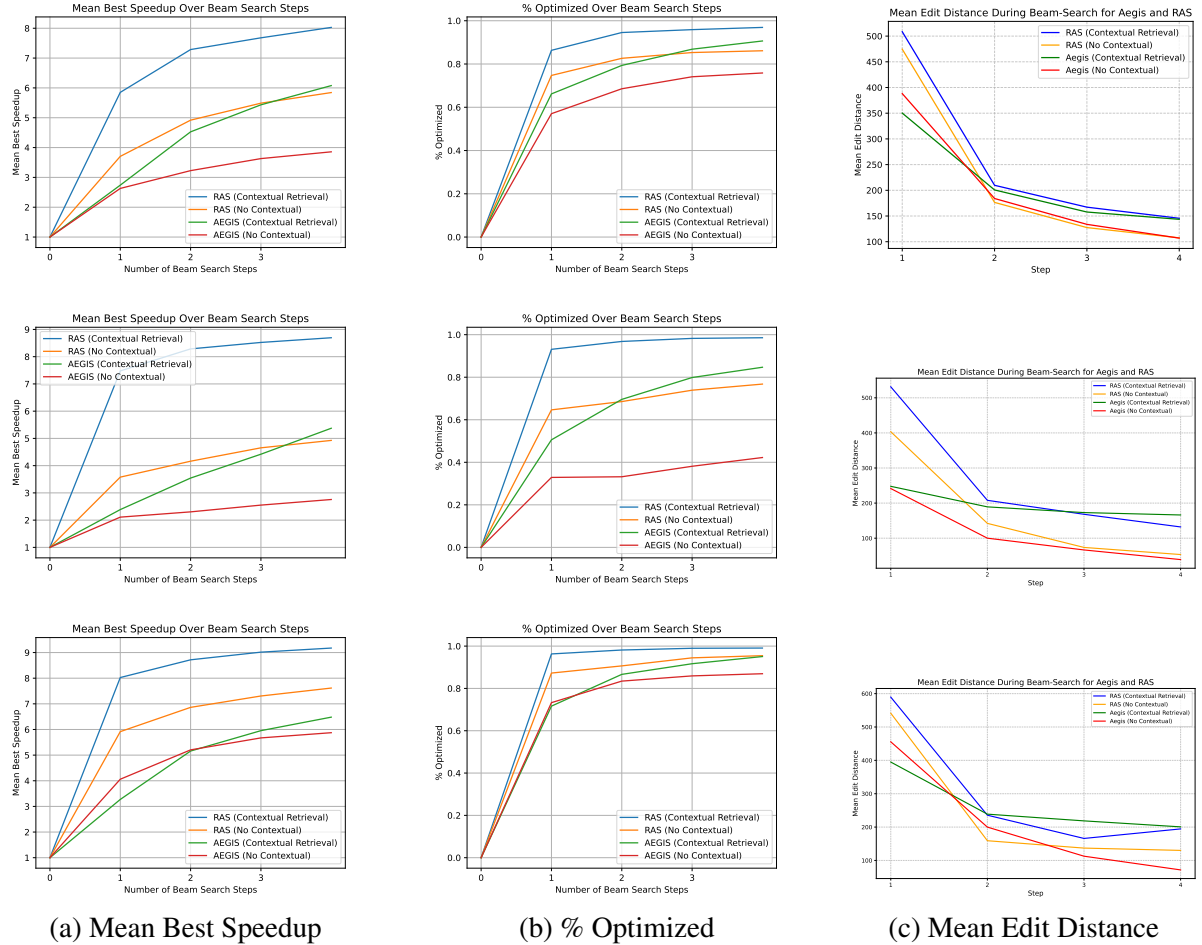


Figure 3: Mean Best Speedup, %Optimized, and Mean Edit Distance across beam search steps on GPT-4o (top), Qwen-3-Coder (middle), and Deepseek 3.2 (bottom) experiments.

dynamic Retrieval” and “Instruct Only” do not perform search). Figure 3 (a) shows results for “Mean Best Speedup”. As can be seen, while the first step of beam search provides the greatest benefit, it continues to benefit all approaches, especially when using contextual retrieval. Since we request the LLM F_{context} to describe the algorithm used for the current best-performing program p_i at each iteration i of the beam search, we hypothesize that F_{context} can update its description to include algorithmic updates made in the previous iteration, thus enabling it to retrieve more relevant examples. We also see greater continuing improvements for AEGIS, likely because atomic edits constrain optimization to change the program more slowly. Additional iterations may help further close the gap between AEGIS and RAS. We provide an example of how AEGIS and RAS both optimize the same program in Appendix E.2. Next, Figure 3 (b) shows results for “% Optimized”. These results converge substantially more quickly, likely

because the first iteration is already enough to get above $1.1\times$ speedup for most programs. Nevertheless, we continue to see gains for our AEGIS approach, again suggesting that continuing search may close the performance gap.

E.2 Comparison Between RAS and AEGIS

In Figures 4 and 5, we show an example of the optimization trajectory taken by each RAS and AEGIS in our GPT-4o experiment on the PIE dataset. As can be seen, RAS concentrates a large number of edits in the first step. In contrast, the edits performed by AEGIS are spread out more evenly across different steps.

E.3 Failure Category Analysis of RAS and AEGIS

In our GPT-4o experiment on the PIE dataset, for each of our two methods, AEGIS and RAS, we construct a set of unoptimized programs. The set contains test set programs for whom the final best

speedup after using the method is less than $1.1\times$. We then study the program descriptions generated by the LLM F_{context} during contextual retrieval. By examining the LLM-generated program descriptions of the test set programs in the unoptimized program set, we can measure the frequency of specific frequently occurring terms. We then compare the frequency of these terms in the unoptimized program set to the overall test set to measure if each method fails disproportionately on problems involving certain algorithms or data structures.

AEGIS appears to struggle with optimizing two primary groups of programs: programs that include dynamic programming algorithms and programs that involve binary search over either sorted lists or trees. Across the complete test set, programs with descriptions including the term “dynamic programming” constitute 51.59% of programs and those mentioning “binary search” constitute 4.32 of programs. AEGIS’s set of unoptimized programs constitutes 9.35% of the entire test set. Programs with descriptions including the term “dynamic programming” constitute 45.05% and those with descriptions mentioning “binary search” constitute 12.09% of this unoptimized set. These results suggest that AEGIS fails disproportionately on binary search problems.

For RAS, the unoptimized program set is 3.08% of the entire test sets. When compared to AEGIS, the percentage of dynamic programming problems in RAS’s unoptimized program set decreases to 16.67%, while the percentage of binary search problems decreases to 6.67%. Additionally, for RAS, we observe a high failure rate for programs whose descriptions mention Kruskal’s algorithm: while such problems constitute 1.03% of the total test set, they constitute 20% of RAS’s unoptimized program set. Only 3.30% of problems in AEGIS’s unoptimized program set mention Kruskal’s algorithm, and we observe that RAS fails on a greater number of problems involving Kruskal’s algorithm as compared to AEGIS.

Thus, we can develop greater insights into their failure modes by analyzing the artifacts produced during RAS and AEGIS. By identifying algorithms and data structures that cause these failures, we can subsequently augment the training dataset which we use for retrieval by targeting specific categories of optimizations for subsequent improvements in performance.

E.4 Mean Edit Distance

We show mean edit distances in Table 5.

E.5 Impact of Code Embedding Models

We also examine whether using specialized code embedding models can reduce the gap between contextual and dynamic retrieval performance. In this experiment, we replace the text-embedding-3-large retrieval model with Codestral-Embed-2505, a specialized code embedding model. We evaluate one iteration of the “No Contextual” ablation for RAS in Table 1 (which uses code embeddings) for one iteration using DeepSeek 3.2 (our best-performing model on PIE). In the first iteration, RAS achieved 8.03 mean best speedup with 96.30% of the test set optimized, the no contextual baseline with text-embedding-3-large achieved 5.92 mean best speedup with 87.26% optimized, and the no contextual baseline with Codestral Embed embeddings achieved 6.57 mean best speedup with 88.39% optimized. Our results show that retrieving programs that utilize similar data structures and algorithms via contextual retrieval leads to much better performance than retrieving programs that share similar code or text embeddings, since competitive programmers often use similarly named variables and operations in very different programs, which may cause code embedding-based retrieval to retrieve less relevant program pairs. Additionally, LLM-generated code is not likely to resemble the optimized programs written by human programmers for competitive programming contests, leading to further difficulty in finding relevant program examples in subsequent search iterations. On the other hand, contextual retrieval abstracts away these differences by retrieving only on the basis of similar data structures or algorithms, leading to the retrieval of more relevant program pairs.

E.6 Mercury Results for Larger Models

We show results of using larger models to optimize programs in the Mercury dataset in the Instruct Only setting in Table 6.

F LLM Usage

LLMs were used to generate the code for executing some of the experiments, which was reviewed by the authors.

Method	GPT-4o	Qwen-3-Coder	DeepSeek 3.2
AEGIS	213.05	194.05	263.24
RAS	257.77	259.87	296.41
AEGIS (No Contextual)	203.24	111.78	209.98
RAS (No Contextual)	221.49	168.20	241.91

Table 5: Comparisons of mean edit distances over steps between AEGIS and RAS on the PIE Benchmark.

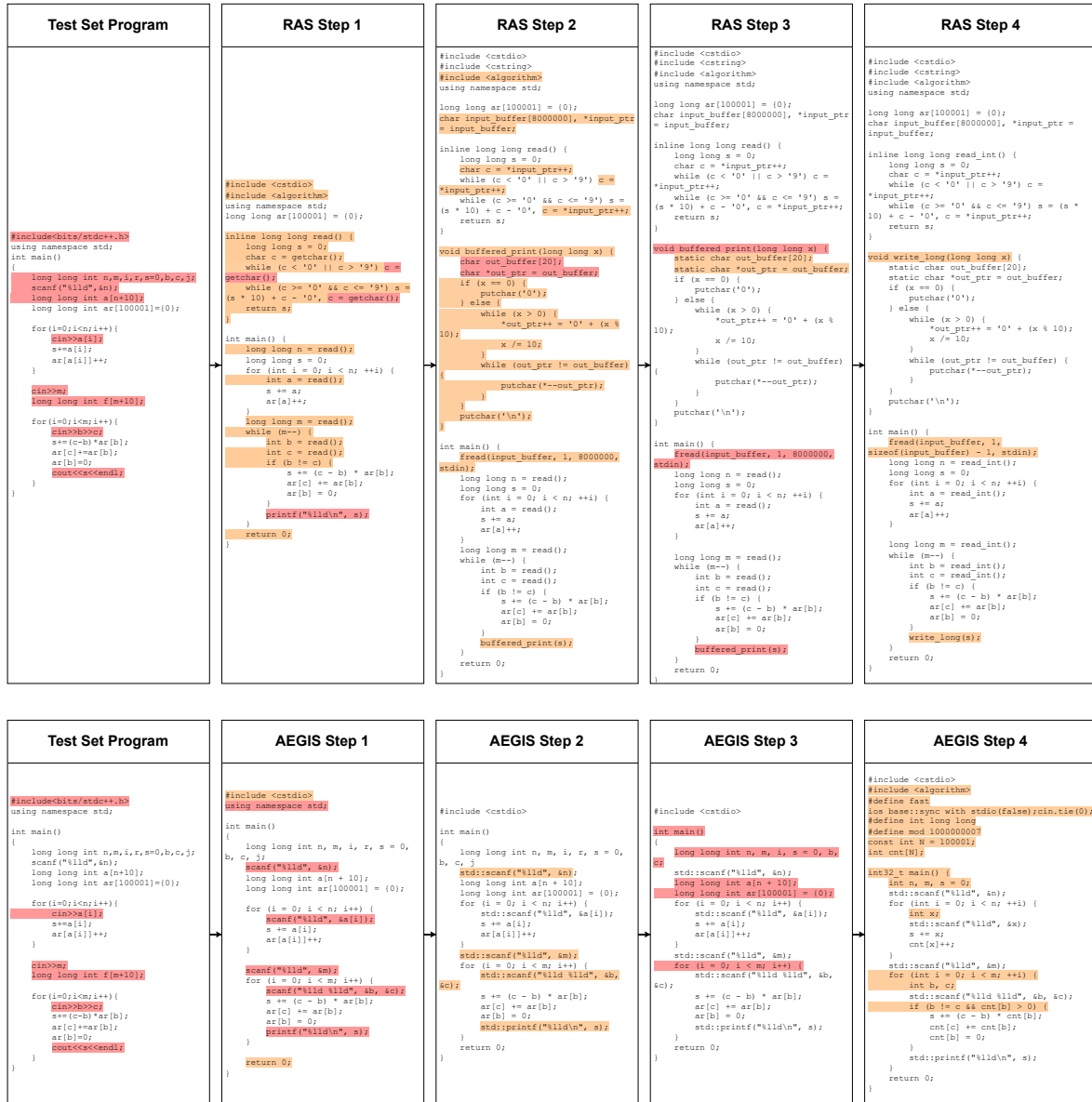


Figure 4: We show a randomly selected example optimization trajectory from our GPT-4o experiments where RAS and AEGIS implement similar optimizations to achieve similar speedups. The final speedup of RAS is $10.06\times$, compared to $9.58\times$ for AEGIS. We have highlighted lines that have changed from the previous step in orange, while lines that change in the next step have been highlighted in red. For reference, the human speedup on this example is $1.8\times$. Here RAS implements an optimization to replace `cin` and `cout` alongside an optimization to ensure that variables are not needlessly updated when `b = c` in Step 1. AEGIS implements a `cin` and `cout` replacement in Step 1 and refines it until Step 3, and then implements the `b!=c` check in Step 4.

Test Set Program	RAS Step 1	RAS Step 2	RAS Step 3	RAS Step 4
<pre> #include <iostream> #include <stdio.h> #include <math> #include <string.h> using namespace std; const int N=100005; int f[N],a[N]; int main() { int n,k; scanf("%d%d",&n,&k); memset(f,0x3f,sizeof(f)); for(int i=1;i<=n;i++) scanf("%d",&a[i]); f[1]=0; for(int i=2;i<=n;i++) for(int j=1;j<=k;j++) if(i>=j) f[i]=min(f[i],f[i-j]+abs(a[i]-a[i-j])); printf("%d\n",f[n]); } </pre>	<pre> #include <stdio.h> #include <stdlib.h> #include <string.h> #include <math.h> #define INF 0x3f3f3f3f int min(int a, int b) { return a < b ? a : b; } int main() { int n, k; scanf("%d%d", &n, &k); int *a = (int *)malloc(n * sizeof(int)); int *f = (int *)malloc(n * sizeof(int)); for (int i = 0; i < n; i++) scanf("%d", &a[i]); f[0] = 0; for (int i = 1; i < n; i++) for (int j = 1; j <= k && i - j >= 0; j++) f[i] = min(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n - 1]); return 0; } </pre>	<pre> #include <stdio.h> #include <stdlib.h> #define INF 0x3f3f3f3f int min(int a, int b) { return a < b ? a : b; } int main() { int n, k; scanf("%d%d", &n, &k); int *a = (int *)malloc(n * sizeof(int)); int *f = (int *)malloc(n * sizeof(int)); for (int i = 0; i < n; i++) scanf("%d", &a[i]); f[0] = 0; for (int i = 1; i < n; i++) for (int j = 1; j <= k && i - j >= 0; j++) f[i] = min(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n - 1]); free(a); free(f); return 0; } </pre>	<pre> #include <stdio.h> #include <stdlib.h> #include <limits.h> int min(int a, int b) { return a < b ? a : b; } int main() { int n, k; scanf("%d%d", &n, &k); int *a = (int *)malloc(n * sizeof(int)); int *f = (int *)malloc(n * sizeof(int)); f[0] = 0; for (int i = 1; i < n; i++) for (int j = 1; j <= k && i - j >= 0; j++) f[i] = min(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n - 1]); free(a); free(f); return 0; } </pre>	<pre> #include <stdio.h> #include <stdlib.h> #include <limits.h> #include <algorithm> int main() { int n, k; scanf("%d%d", &n, &k); int *a = (int *)malloc(n * sizeof(int)); int *f = (int *)malloc(n * sizeof(int)); f[0] = 0; for (int i = 1; i < n; i++) for (int j = 1; j <= k && i - j >= 0; j++) f[i] = std::min(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n - 1]); free(a); free(f); return 0; } </pre>
<pre> #include <iostream> #include <stdio.h> #include <math> #include <string.h> using namespace std; const int N=100005; int f[N],a[N]; int main() { int n,k; scanf("%d%d",&n,&k); memset(f,0x3f,sizeof(f)); for(int i=1;i<=n;i++) scanf("%d",&a[i]); f[1]=0; for(int i=2;i<=n;i++) for(int j=1;j<=k;j++) if(i>=j) f[i]=min(f[i],f[i-j]+abs(a[i]-a[i-j])); printf("%d\n",f[n]); } </pre>	<pre> #include <stdio.h> #include <math.h> #include <string.h> using namespace std; const int N = 100005; int f[N], a[N]; int main() { int n, k; scanf("%d%d", &n, &k); memset(f, 0x3f, sizeof(f)); for (int i = 1; i <= n; i++) scanf("%d", &a[i]); f[1] = 0; for (int i = 2; i <= n; i++) for (int j = 1; j <= k; j++) if (i >= j) f[i] = min(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n]); return 0; } </pre>	<pre> #include <stdio.h> #include <math.h> #include <string.h> using namespace std; const int N = 100005; int f[N], a[N]; int main() { int n, k; scanf("%d%d", &n, &k); memset(f, 0x3f, sizeof(f)); for (int i = 1; i <= n; i++) scanf("%d", &a[i]); f[1] = 0; for (int i = 2; i <= n; i++) for (int j = 1; j <= k; j++) if (i >= j) f[i] = min(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n]); return 0; } </pre>	<pre> #include <stdio.h> #include <math> #include <string.h> #include <algorithm> using namespace std; const int N = 100005; int f[N], a[N]; inline void updateMinimum(int &target, int source) { target = min(target, source); } int main() { int n, k; scanf("%d%d", &n, &k); memset(f, 0x3f, sizeof(f)); for (int i = 1; i <= n; i++) scanf("%d", &a[i]); f[1] = 0; // Optimized loop to only iterate over relevant parts for (int i = 2; i <= n; i++) for (int j = min(i - 1, k); j >= 1; j--) updateMinimum(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n]); return 0; } </pre>	<pre> #include <stdio.h> #include <math> #include <string.h> #include <algorithm> using namespace std; const int N = 100005; int f[N], a[N]; inline void updateMinimum(int &target, int source) { target = min(target, source); } int main() { int n, k; scanf("%d%d", &n, &k); memset(f, 0x3f, sizeof(f)); for (int i = 1; i <= n; i++) scanf("%d", &a[i]); f[1] = 0; // Optimized loop to only iterate over relevant parts for (int i = 2; i <= n; i++) for (int j = min(i - 1, k); j >= 1; j--) updateMinimum(f[i], f[i - j] + abs(a[i] - a[i - j])); printf("%d\n", f[n]); return 0; } </pre>

Figure 5: We show a randomly selected example optimization trajectory from our GPT-4o experiments where RAS significantly outperforms AEGIS. Here, we demonstrate the improvements made at each step of RAS vs. AEGIS. The final speedup of RAS on this example is $7.34\times$, compared to $2.35\times$ for AEGIS. We have highlighted lines that have changed from the previous step in orange, while lines that change in the next step have been highlighted in red. For reference, the human speedup on this example is $1.37\times$.

Method	Beyond@1	Pass@1
Instruct Only (Gemma3-12B) ($h = 16$)	88.76	99.61
Instruct Only (Qwen3-30B) ($h = 8$)	91.02	99.22
Instruct Only (Qwen-QwQ-32B) ($h = 8$)	93.13	99.61

Table 6: RAS Experiments on Mercury on larger models.