

SemOpt: LLM-Driven Code Optimization via Rule-Based Analysis

Yuwei Zhao

Key Laboratory of High Confidence
Software Technologies (Peking
University), Ministry of Education;
School of Computer Science, Peking
University
Beijing, China
zhaoyuwei@stu.pku.edu.cn

Yuan-An Xiao

Key Laboratory of High Confidence
Software Technologies (Peking
University), Ministry of Education;
School of Computer Science, Peking
University
Beijing, China
xiaoyuanan@pku.edu.cn

Qianyu Xiao

Key Laboratory of High Confidence
Software Technologies (Peking
University), Ministry of Education;
School of Computer Science, Peking
University
Beijing, China
2200012932@stu.pku.edu.cn

Zhao Zhang

Key Laboratory of High Confidence
Software Technologies (Peking
University), Ministry of Education;
School of Computer Science, Peking
University
Beijing, China
zhangzhao2019@pku.edu.cn

Yingfei Xiong*

Key Laboratory of High Confidence
Software Technologies (Peking
University), Ministry of Education;
School of Computer Science, Peking
University
Beijing, China
xiongyf@pku.edu.cn

Abstract

Automated code optimization aims to improve performance in programs by refactoring code, and recent studies focus on utilizing LLMs for the optimization. Typical existing approaches mine optimization commits from open-source codebases to construct a large-scale knowledge base, then employ information retrieval techniques such as BM25 to retrieve relevant optimization examples for hotspot code locations, thereby guiding LLMs to optimize these hotspots. However, since semantically equivalent optimizations can manifest in syntactically dissimilar code snippets, current retrieval methods often fail to identify pertinent examples, leading to suboptimal optimization performance. This limitation significantly reduces the effectiveness of existing optimization approaches.

To address these limitations, we propose SemOpt, a novel framework that leverages static program analysis to precisely identify optimizable code segments, retrieve the corresponding optimization strategies, and generate the optimized results. SemOpt consists of three key components: (1) A strategy library builder that extracts and clusters optimization strategies from real-world code modifications. (2) A rule generator that generates Semgrep static analysis rules to capture the condition of applying the optimization strategy. (3) An optimizer that utilizes the strategy library to generate optimized code results. All the three components are powered by LLMs.

On our benchmark containing 151 optimization tasks, SemOpt demonstrates its effectiveness under different LLMs by increasing the number of successful optimizations by 1.38 to 28 times compared to the baseline. Moreover, on popular large-scale C/C++ projects, it can improve individual performance metrics by 5.04% to 218.07%, demonstrating its practical utility.

1 Introduction

Computational efficiency is a key factor in software that influences its quality [12, 15]. Inefficient code segments can lead to increased system runtime, waste of computational resources, and a degraded user experience [23, 33]. Existing research indicates that optimization opportunities are prevalent in software and may be costly and tedious for human developers to detect and optimize [5, 23]. Therefore, automated code optimization, which involves automatically refactoring code to improve performance metrics, has attracted considerable attention from researchers in recent years.

Early research on code optimization primarily focused on rule-based approaches, which address specific types of inefficiencies [24, 33]. However, these methods rely heavily on predefined rules crafted by experts, making them labor-intensive and lacking scalability, and ultimately limiting their coverage to a narrow range of problems [2].

With the advancement of deep learning technologies, particularly the emergence of large language models (LLMs), there has been a surge of research inspired by these developments and focused on LLM-based approaches [10, 20, 22, 25, 26, 28, 32, 37, 46–48]. Compared to rule-based approaches, LLM-based approaches are able to utilize the code understanding and generation capability of LLMs, generalizing to a large variety of optimization scenarios and optimization strategies without human intervention, leading to many optimizations that are infeasible in rule-based approaches.

The Challenge of LLM-based Optimization. A direct approach is to give a code snippet to LLM and ask the LLM to optimize the code. However, as observed in multiple existing studies [13, 43], this approach is ineffective, possibly due to the many possible directions of optimizing the code. A more effective approach is to identify the optimization strategy (e.g., replacing a linked list with an array if frequent random access is needed) to be applied in the code, and include one or a few existing optimization examples applying this strategy in the prompt for few-shot learning, such that the LLM

*Corresponding author

learns the optimization strategy from the examples and applies it to the current code.

However, applying the above few-shot learning to optimize a large software project is not easy. On the one hand, to get many optimization examples representing various optimization strategies, existing approaches [13, 16, 43] usually mine optimization commits from many open source projects, forming *a large space of example commits*. On the other hand, since LLM can process code only to a limited length, we have to divide the project code into multiple small chunks for LLM to process, forming *a large space of code locations*. Since it is difficult to know which strategy can be applied to which code locations in advance, a trivial approach is to try all combinations of the commits and the code locations, which is infeasible in practice due to the huge number of combinations.

To address this problem, existing approaches [13, 16, 43] have proposed using information retrieval techniques such as BM25 [41] to retrieve the most relevant optimization commit for a code location. In this way, we can enumerate the hotspot locations (which can be identified by profiling and are often small in number) and retrieve the most relevant optimization commits for these locations, effectively addressing the problem of large spaces. However, as our evaluation will show later, since the same optimization strategy can be applied to significantly dissimilar code snippets, even the best-performing retrieval techniques in the existing studies often cannot retrieve relevant commits, leading to low performance in practice.

Our Approach. To address the above challenge, our main idea is that, instead of retrieving a commit suitable for the current code location, we generate a symbolic pattern-matching rule from a commit to match the code locations where the optimization strategy represented by the commit can be applied. More concretely, given an optimization commit, we design an LLM agent to summarize the optimization strategy used in this commit, and then generate a Semgrep [42] rule to match the code locations that are suitable to be optimized using this strategy. Semgrep is a static analysis tool that allows the user to write customized rules to efficiently match code locations in a large project. In this way, we utilize the analytic power of the LLM to understand the optimization strategy and extract symbolic rules to more precisely capture the application condition of the strategy, avoiding the imprecision from information retrieval techniques.

With the above method, we can efficiently explore the space of code locations, but we still need to enumerate each commit in the large space of commits to generate and execute the rule extracted from this commit. To reduce this burden, we notice that many commits collected are of the same optimization strategies, and propose a clustering method to cluster the examples based on the strategies summarized by the LLM. In this way, each cluster ideally contains only commits for one strategy. Then, we produce only a few Semgrep rules from a cluster, effectively reducing the burden of enumerating the large space of examples.

Finally, we materialize this approach into a tool called **SemOpt** (for Semgrep + Optimization), as shown in Figure 1. It comprises three key components:

- (1) A **strategy library builder** that mines optimization examples from code commits, generates their strategy descriptions by an LLM, and clusters the examples of the same strategy;
- (2) A **rule generator** that generates Semgrep static analysis rules for each cluster to capture the condition of applying the optimization strategy;
- (3) An **optimizer** that applies Semgrep rules on source code to detect optimizable code locations, and then uses an LLM to generate optimized code for each location.

Evaluation Result. To evaluate the effectiveness of SemOpt, we constructed a benchmark comprising 151 C/C++ code optimization problems, covering a wide range of optimization strategies. We then compared the performance of SemOpt on this benchmark with two baseline methods using three popular LLMs: DeepSeek-V3, GPT-4.1, and Gemini-2.5-pro. The experimental results show that SemOpt achieves a significant improvement in producing correct optimization outcomes, reaching 37.5% to 27x more successful optimizations than the baselines under different LLMs.

To evaluate the real-world performance of SemOpt, we further applied SemOpt to five popular open-source C/C++ projects. Across these projects, SemOpt achieves a maximum performance improvement of 5.04% to 218.07% in a single performance test case, and delivers an average improvement of 1.68% to 10.40% across all performance test cases. This demonstrates that SemOpt is capable of automatically optimizing large-scale code projects in the real world.

Our Contributions. Our contributions are as follows.

- The first LLM-based automated code optimization approach that leverages the static program analysis tool (Section 3), which significantly improves the efficiency of identifying optimizable code snippets and selecting optimization strategy.
- A workflow combining LLM and clustering algorithms to summarize and merge similar optimization strategies (Section 3), effectively reducing the size of the strategy library.
- Extensive experiments on a benchmark comprising 151 code optimization problems (Section 4), demonstrating that SemOpt is effective in optimizing code across different LLMs. Furthermore, it can automatically optimize large-scale projects in the real world (Section 5), highlighting its practical value.

2 Motivating Example

In this section, we provide an overview of the library constructed by SemOpt and how we use this library through a motivating example in Figure 2. How we construct the library will be explained later in Section 3.

The code before optimization contains an if statement in a loop, which can be optimized. The condition to be evaluated in this if statement consists of two sub-conditions connected by the && operator. According to the short-circuit evaluation mechanism of Boolean expressions, if the result of the first sub-condition is false, the entire condition will be false, and thus, there is no need to evaluate the second sub-condition before proceeding to the corresponding branch. Therefore, we can place the sub-condition that is relatively easier to evaluate at the beginning of the if statement, and the more complex sub-condition later. In this way, if the result of the earlier sub-condition is false, the program can directly

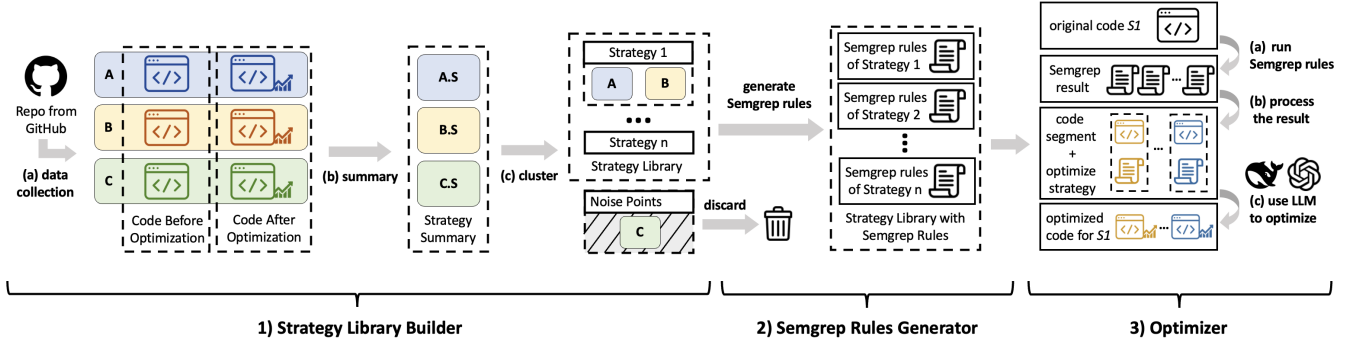


Figure 1: The overview of SemOpt.

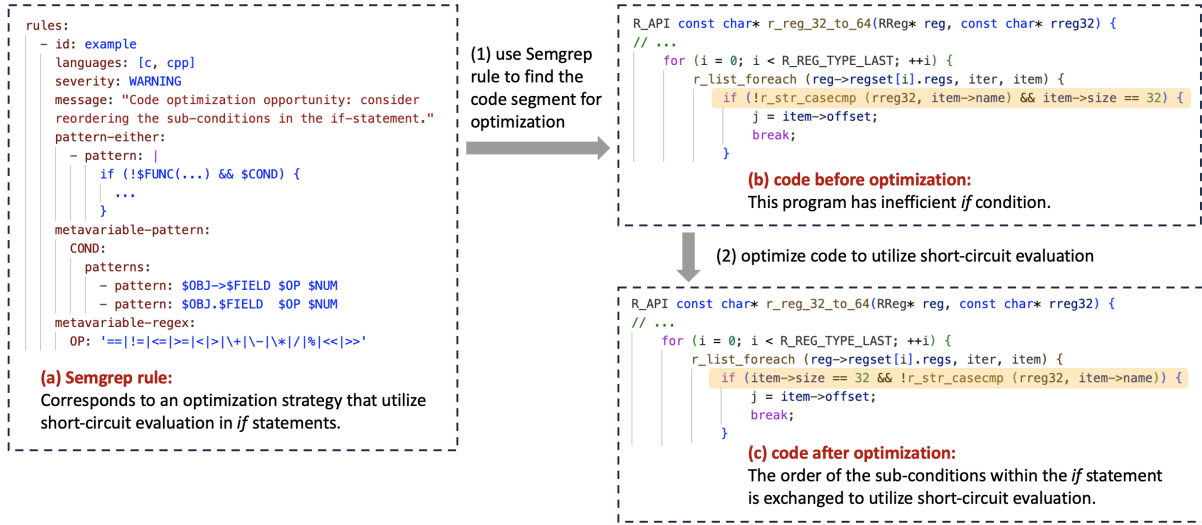


Figure 2: The motivating example that leverages short-circuit evaluation to optimize the performance.

proceed to the corresponding branch without evaluating the more complex part, thus improving the code execution efficiency. In the *if* statement shown in Figure 2b, the evaluation of the first sub-condition is generally more complex than that of the second. Therefore, we can improve the runtime efficiency of the code by exchanging the order of the two sub-conditions.

Figure 1 illustrates how SemOpt generates optimizations:

First, we build a library containing a large number of different code optimization strategies. Each optimization strategy is associated with corresponding Semgrep rules for static program analysis, generated automatically from commits by an LLM. For example, there is a Semgrep rule from the strategy library in Figure 2a. It corresponds to an optimization strategy that leverages short-circuit evaluation in *if* statements. Specifically, this rule matches a class of *if* statements containing two sub-conditions connected by an *&&* operator, with the requirement that the first sub-condition negates the result of a function call, and the second sub-condition involves accessing the value of a particular field in an object, which is then operated on with an arbitrary number. If an *if* statement can be identified by this Semgrep rule, it is advisable to consider swapping the order of the two sub-conditions within the *if* statement

to leverage short-circuit evaluation for improved code execution efficiency.

Then, whenever we have a target project to optimize, we run the Semgrep rules from the strategy library to determine at what locations the current optimization strategy can be applied. In Figure 2, if we run the Semgrep rule on the code before optimization, we can find that the corresponding optimization strategy can be applied, and the precise location of the *if* statement is identified.

After identifying an applicable optimization strategy, we construct a corresponding optimization prompt, as shown in Figure 3. The prompt mainly consists of four parts:

- (1) The complete code that needs to be optimized with line number annotations
- (2) The location information of the code to be optimized represented by the line number range.
- (3) The optimization strategy to be used.
- (4) A requirement for the model to apply the optimization strategy to the given code segment and provide the complete content of the optimized code.

Please optimize the following C/C++ function based on the identified code segment and reference optimization strategies.

FUNCTION WITH LINE NUMBERS:

```
...
line 10: if (!r_strcasecmp (rreg32, item->name) && item->size == 32) {
line 11:   j = item->offset;
line 12:   break;
line 13: }
...
```

TARGET CODE SEGMENT TO OPTIMIZE:

Lines 10 to 13

REFERENCE OPTIMIZATION STRATEGIES:

Reorder sub-conditions in if statements with multiple 'and'-connected conditions to place simpler or less expensive checks before more complex ones for better short-circuit evaluation.

...

REQUIREMENTS:

1. Optimize the given code segment using the provided strategies.
2. Output one code block with the complete optimized function.

Figure 3: The prompt for generating optimization results.

Figure 2c shows the optimization result generated by the LLM based on the prompt in Figure 3, in which the order of the sub-conditions in the if statement is swapped to utilize short-circuit evaluation and improve code execution speed. This optimization exactly matches that provided by the developer.

Note that the rules in our library are not necessarily precise (e.g., in this example, a function call may not necessarily lead to a heavier evaluation than the second condition) or complete (e.g., the example optimization strategy can be applied even if there is no negation), since they are generated by an LLM. To address these issues, we further employ two methods: 1) to address incompleteness, we generate multiple rules per strategy, where the rules together forming a more complete set; 2) to address imprecision, we sort the locations by the number of rules matching them, where the location matched by more rules are more likely to be true optimization opportunities. The details will be discussed in Section 3. Furthermore, the precision issue is also addressed by the later optimization generation step, as the LLM will fail to generate an optimized code snippet if the code is unsuitable to be optimized using this strategy.

3 Approach

In this section, we discuss the details of three components in SemOpt: a library builder, a rule generator, and an optimizer. The overall pipeline of our approach is illustrated in Figure 1.

3.1 Building the Strategy Library

The first step of our methodology is to construct a code optimization strategy library, which contains textual descriptions and representative code examples for each optimization strategy. Subsequently, for each optimization strategy, we generate corresponding Sengrep rules (Section 3.2), thereby establishing a comprehensive strategy library. The static program analysis rules in this strategy library can be used to identify optimization opportunities, which then guide the LLM in applying the corresponding strategies and producing optimized code. The following describes how relevant data is collected to construct the strategy library, as well as how clustering is

employed to retain effective information and reduce the size of the library.

3.1.1 Data Collection. The first step of our approach is to collect a large set of optimization commits. Our current implementation focuses on C and C++ programming languages. Concretely, we first collected 30342 codebases from GitHub with at least 100 stars, whose primary language is C or C++, and which have had commit activity within the past five years. We crawled the commit histories of the main branches of these codebases, ensuring that for each commit, we collected the commit message, the complete code before the modification, the complete code after the modification, and the diff file that records the specific changes introduced by the commit. In total, we obtained 108,039,931 commits. In this work, we focus solely on code optimizations within individual functions. Therefore, we first filter for commits that modify only a single function in C or C++. Next, we use a set of optimization-related keywords to match commit messages, and further leverage an LLM in conjunction with the corresponding diff files to verify the nature of the changes. Besides, we perform deduplication by analyzing both commit messages and code modifications. Through this process, we ultimately identify 35,668 commits that primarily implement code optimizations.

3.1.2 Summarization of Optimization Strategies. This part is mainly divided into two modules.

First, we ask an LLM to generate summaries of the optimization strategy used in each commit. Concretely, for each code optimization-related commit, we collect detailed information, including the name of the codebase, the commit message, the name of the modified function, and the complete diff file. Then, we include all this information in the prompt and use LLMs to generate a one-sentence summary of the corresponding optimization strategy. To reduce variability in the generation process and improve the reliability of the results, we independently generate m summaries for each commit. In our implementation, we set m as 3.

Second, we select the best summary among the generated summaries for each commit. Specifically, we first use a pre-trained model to encode each summary into a high-dimensional semantic vector. Next, we calculate the pairwise cosine similarities to construct a similarity matrix. Finally, we compute the average similarity of each summary to the others and select the one with the highest average as the final optimization strategy summary for that commit.

3.1.3 Strategy integration and selection. First, as in previous steps, we utilize a pre-trained model to convert the strategy summary corresponding to each commit into a high-dimensional semantic vector.

Then, we apply an unsupervised clustering algorithm, DBSCAN [8], based on cosine similarity to all semantic vectors. By setting the hyperparameter eps as 0.89 in this algorithm, we control the minimum similarity within each cluster, ensuring that the strategy summaries grouped together exhibit a high degree of semantic similarity in the vector space. We consider each cluster to represent one optimization strategy.

After clustering, some clusters are very small, and we remove these clusters because they represent strategies that appeared infrequently across the codebases and thus are unlikely to be applicable again. Removing them could improve the quality of the library and the efficiency of the whole system.

3.2 Automatic Generation of Analysis Rules

After constructing the basic optimization strategy codebase using the method described in Section 3.1, we need to generate corresponding Semgrep rules for each category of optimization strategy in order to identify potential optimization opportunities associated with each strategy.

First, as we have seen in the motivation example section, the Semgrep rules may not be complete. To overcome this problem, we randomly sample up to n commits from a cluster and generate a rule from each sampled commit. Since the rule generated from different commits may capture different scenarios where the strategy can be applied, the rule set would be relatively complete when the n is large enough. In our implementation, we set n as 10.

Second, for each commit, we have developed a dedicated agent that automatically generates Semgrep rules by iteratively invoking an LLM and incorporating its feedback. The main process is as follows:

- **Understand the optimization strategy:** We provide the diff file of the commit to the LLM, and ask it to thoroughly analyze and explain the optimization strategy used in this commit. At this stage, the focus is solely on understanding; no Semgrep rules are generated.
- **Generate an initial rule:** Based on the previous analysis, the LLM produces a candidate Semgrep rule. This rule should be designed to detect similar optimization opportunities and must strictly adhere to Semgrep’s syntax requirements.
- **Validate and iteratively refine the rule:** The generated Semgrep rule is executed on the pre-commit version of the code, where the commit is used for rule generation. If an error occurs, the error information and context, together with the current rule, are automatically fed back to the LLM for targeted revision. This process repeats until the rule executes successfully or a predefined maximum number of iterations is reached.

Additionally, based on practical experiments and insights from existing work [14], we observed that simply increasing the maximum number of iterations yields limited improvement in the final results. Instead, generating multiple independent Semgrep rules for each commit is more likely to produce desirable outcomes. Therefore, we configure the process to generate Semgrep rules 5 times independently for each commit, with each attempt allowing a maximum of 7 iterations.

3.3 Strategy Library-Based Code Optimization

After generating Semgrep rules for each strategy in the strategy library using the method described in Section 3.2, this strategy library can then be utilized to optimize code. The automated code optimization framework presented in this section is primarily divided into three components, which will be introduced in detail below.

First, after obtaining a piece of code that needs to be optimized, we use the Semgrep rules corresponding to all the policies in the strategy library to scan the code and record the results of the scan. Each scan result mainly contains two types of information: the location information of the code fragments matched by the Semgrep rules, and the optimization strategy information corresponding to those Semgrep rules. Since Semgrep is a static program analysis tool, it can analyze incomplete code, such as individual files or functions. Furthermore, it operates with high speed, typically completing all scans on files with several hundred or even over a thousand lines of code in about 1 second.

Next, we need to further process the scan results. Since each type of strategy may correspond to multiple Semgrep rules, the same location may be matched by multiple rules. As mentioned in Section 2, since the rules may be imprecise, we can use the number of rules matching a location as an indicator to show how likely the location is a true optimization opportunity. We record both the number of times each location is detected and sort the locations by the number of times they are detected. We then select only the top n fragments per function and their corresponding policies for optimization, where n is set to 25 in our implementation.

Finally, for each code fragment and its corresponding optimization strategy, we aggregate all relevant information and invoke the LLM to generate the optimized result. As we have seen in Figure 3, the prompt includes information such as the location of the code fragment to be optimized, the optimization strategy to be applied, and the complete code requiring optimization. For further details, please refer to Section 2.

4 Empirical Evaluation

We evaluate SemOpt with the following research questions:

- RQ1:** Does SemOpt produce more successful optimizations than baselines?
- RQ2:** How important is each component of SemOpt?
- RQ3:** Do optimization strategies in SemOpt and baselines generalize across codebases?

4.1 Experimental Setup

4.1.1 Benchmark. In this work, we constructed a benchmark to evaluate code optimization tools. We collected 2,953,660 commits from the main branches of the 100 most-starred C/C++ codebases on GitHub. We first filtered commits to retain only those modifying a single C/C++ function. Using optimization-related keywords to match commit messages, we then employed an LLM to analyze the corresponding diff files for verification. After deduplication based on commit messages and code changes, we obtained 2,529 commits that primarily implement code optimizations, from which we randomly sampled 151 commits to construct our benchmark dataset.

Since the benchmark is constructed from historical commits on GitHub, it overlaps with the data used by baselines and SemOpt. This includes the knowledge base employed by RAPGen and RAG, as well as the strategy library of SemOpt. To address the data leakage problem, during the evaluation, we exclude information exactly matching the commit or the code of the current task from the data

used in baselines and SemOpt. Only the remaining data is available for optimization, thereby mitigating the problem of data leakage.

Each entry in the benchmark contains the codebase name, the commit hash, and the complete function before and after the commit. The function before commit is the input for code optimization tools, and the function after commit is used to calculate the ground truth patch by the developer, so that we can assess the correctness of the optimization results.

Note that we constructed a new benchmark instead of reusing existing ones because no existing benchmarks suit our needs as far as we are aware. For example, the benchmark used by RAPGen [16] consists of optimization tasks for C# code, whereas our tool is implemented to optimize only C/C++ code. The PIE benchmark [43] derived from CodeNET [38] is typically small in scale, whereas we aim to evaluate optimization capabilities on large-scale codebases in real-world scenarios.

4.1.2 Baselines. We compared SemOpt with three baseline approaches, as follows.

Direct Prompting: This baseline uses a static prompt, requiring the LLM to directly provide the optimized results.

Retrieval Augmented Generation (RAG): This baseline implements the standard RAG process. The implementation follows previous work [14]. We first gather real-world code optimization-related commits to construct a comprehensive knowledge base. The collection process follows the same approach as in Section 3.1. For each piece of code to be optimized, we follow Gao et al. [14] to use the BM25 algorithm [41] to retrieve the four most similar code examples from the knowledge base, sort them in ascending order of similarity, and then include them in the prompt along with the current code to be optimized. The LLM is then instructed to generate an optimized version of the code.

RAPGen: We compared against RAPGen [16], the state-of-the-art approach for large-scale code optimization as we are aware. Since RAPGen’s source code is not publicly available, we re-implemented the tool according to the descriptions in its paper and adapted it for code optimization tasks on C/C++ code. Since the RAPGen approach does not identify the location to be optimized, it is necessary to specify the line number of the code to be optimized in the input. The RAPGen then uses the corresponding code line to match optimization strategies within its knowledge base. If multiple strategies are found, the tool calculates similarity scores to identify the most suitable optimization strategy, which is then applied to the code. In our implementation of the RAPGen tool, it enumerates each line of the code and attempts to match optimization strategies from the knowledge base for each line. After collecting all potential strategies, we then use RAPGen’s similarity calculation method to identify the final strategy to be applied.

Additionally, we also implemented an enhanced version of the RAPGen tool (RAPGen+), which additionally provides the location information of the modified code segments from the original commit as input. Subsequently, only the code within these segments is used to match optimization strategies in the knowledge base, after which the similarity is calculated to determine the final strategy to be applied. This setup represents an ideal upper bound of the

performance of RAPGen if the optimization location is precisely known in advance (which is not true for all other baselines).

4.1.3 Evaluated LLMs. Our evaluation of SemOpt and the baselines was performed on three LLMs, i.e., DeepSeek-V3 (version 0324) [27], GPT-4.1 (version 2025-04-14) [34], and Gemini-2.5-Pro [6], via their official APIs. Our choice of LLMs covers both open-source and proprietary LLMs with decent coding and reasoning capabilities. Due to cost constraints, we evaluated Gemini-2.5-Pro only for direct prompting, RAG, and SemOpt on a randomly selected subset of 40 code optimization problems. For the ablation studies in RQ3, we exclusively used DeepSeek-V3.

4.1.4 Metrics. To evaluate the correctness of the optimized code, we employ two metrics, following existing work [16].

- **Exact Match (EM):** We normalize both the ground-truth code by the developer and the optimization results generated by each tool by removing comments, indentation, and whitespace characters. We then compare the normalized code strings to check whether they are identical.
- **Semantic Equivalence (SemEqv):** We retrieve the automatically generated patch and the developer’s ground-truth patch. The authors then manually compare the two patches to determine whether they are semantically equivalent.

For each optimization task, if at least one solution generated by a given method meets the evaluation criteria, we consider the method to have successfully solved the task.

4.1.5 Implementation Details. For all LLMs, we follow prior work [3, 14, 31] and set the temperature to 0. Each optimization task is repeated three times, and we report the average number for all metrics to mitigate randomness. To build the strategy library (Section 3.1), we use DeepSeek-V3 [27] to summarize the optimization strategy, and all-MiniLM-L6-v2 [40, 45] to encode the summary into a high-dimensional semantic vector.

4.2 RQ1: Comparison with Baselines

To evaluate the effectiveness of SemOpt in code optimization, we compared it with RAPGen, its enhanced version RAPGen+, and two representative prompt engineering techniques – retrieval-augmented generation (RAG) and direct prompting – across the three LLMs. Table 1 presents the performance of each tool on the benchmark of 151 code optimization tasks.

4.2.1 Comparison to RAPGen. As shown in Table 1, SemOpt achieves notable improvements over RAPGen on all LLMs. Under the Exact Match metric, RAPGen successfully optimized 0 and 1 cases, and SemOpt achieved 42 and 28 cases. Under the Semantic Equivalence metric, the number of successful optimizations achieved by SemOpt is 21.33 and 22 times higher than that of RAPGen. All results are based on evaluations with DeepSeek-V3 and GPT-4.1 [27, 34]. These results demonstrate the effectiveness of SemOpt for code optimization tasks.

We studied the reasons behind the optimization failures of RAPGen, which can be summarized into the following three main aspects.

Limited Strategy: RAPGen is limited to a single optimization strategy—modifications on individual APIs. However, only about

Table 1: The performance comparison of baselines and SemOpt on different LLMs.

Approach	DeepSeek-V3		GPT-4.1		Gemini-2.5-Pro (on 40 problems)	
	EM	SemEqv	EM	SemEqv	EM	SemEqv
RAPGen	0 (0.0%)	3 (2.0%)	1 (0.7%)	2 (1.3%)	-	-
RAPGen+	3 (2.0%)	6 (4.0%)	3 (2.0%)	7 (4.6%)	-	-
Direct	2 (1.3%)	9 (6.0%)	7 (4.6%)	13 (8.6%)	0 (0.0%)	3 (7.5 %)
RAG	25 (16.6%)	36 (23.8%)	17 (11.3%)	32 (21.2%)	6 (15.0%)	9 (22.5%)
SemOpt	42 (27.8%)	64 (42.4%)	28 (18.5%)	44 (29.1%)	12 (30.0%)	18 (45.0%)
SemOpt + RAG	49 (32.5%)	75 (49.7%)	37 (24.5%)	58 (38.4%)	16 (40.0%)	21 (52.5%)

```
// Origin Code
if (read_only == false && column_families_not_found.size() > 0)
// Abstract Code
if (<PLACEHOLDER> == false && <PLACEHOLDER>.size() > 0)
// Optimization Strategy: Replace size API with empty API
// Optimized Code
if (read_only == false && !column_families_not_found.empty())
```

(a) The code requiring optimization.

```
// Strategies from Library of RAPGen.
// All of them are "Replace size API with empty API".

// Strategy 1
// Origin Code
return d->data.size() == 0;
// Abstract Code
return <PLACEHOLDER>.size() == 0;

// Strategy 2
// Origin Code
valid_ = valid_ && primary_key_ids_.size() > 0;
// Abstract Code
<PLACEHOLDER> = <PLACEHOLDER> && <PLACEHOLDER>.size() > 0;

// Strategy 3
// Origin Code
if (process->m_req_queue.size() == 0)
// Abstract Code
if (<PLACEHOLDER>.size() == 0)
```

(b) Optimization strategies from RAPGen.

```
rules:
- id: prefer-empty-over-size-check
  message: |
    Prefer using `empty()` instead of `size() > 0` for container emptiness
    checks.
    `empty()` is more readable and consistently O(1) for all standard
    containers.
  languages: [cpp, c]
  severity: INFO
  patterns:
  - pattern: $CONTAINER.size() > 0
  - pattern-not: $CONTAINER.empty()
  fix: $CONTAINER.empty()
```

(c) The Semgrep rule from SemOpt.**Figure 4: Code optimization example for RQ1.**

23 optimization tasks in the benchmark conform to this pattern, which fundamentally restricts the applicability of RAPGen on this benchmark.

Simplistic Strategy Matching Mechanism: RAPGen requires that the abstracted code structurally matches the entries in its strategy library exactly, which may result in the omission of effective strategies. Its abstraction approach replaces project-specific identifiers with placeholders, and stores both the source code before optimization and its corresponding abstracted result in the strategy library. A successful match between the code to be optimized and an entry in the strategy library requires that their abstracted representations be exactly identical. As shown in Figure 4, although several optimization strategies in the library are consistent with the one depicted in Figure 4a, the abstracted result of the code to be optimized does not exactly match the abstracted result of any entry in the library, failing to match a strategy. This issue severely limits the optimization capabilities of RAPGen.

Lack of Localization Capability: RAPGen relies heavily on the input to specify the exact lines of code to be optimized. However, our benchmark does not provide such localization information, forcing RAPGen to attempt to match every line in the input code against its strategy library. This process can introduce a large number of incorrect optimization strategies, thereby negatively affecting its overall optimization performance.

4.2.2 Comparison to RAPGen+. To evaluate whether RAPGen would perform better without the problem of localization, we implemented an enhanced version, RAPGen+, as described in Section 4.1. We explicitly provide the location of the code segments modified in the original commit as additional input to RAPGen+, emulating an ideal condition of precise localization. Subsequently, only the code within these segments is used to match optimization strategies, after which similarity is computed to determine the final strategy to apply.

As shown in Table 1, RAPGen+ demonstrates a slight improvement over RAPGen, with the number of successful optimizations increasing by 2 and 5 cases. However, the number of successful optimizations achieved by SemOpt is still 6.29 and 14 times greater than that of RAPGen+. This indicates that even with additional correct localization information, RAPGen+ still performs significantly worse than SemOpt.

4.2.3 Comparison to Prompt Engineering and RAG. As shown in Table 1, the number of successful optimizations achieved by SemOpt is much higher than the direct prompt engineering (Direct) and the RAG baseline. For the Exact Match metric, SemOpt has 4 to

Table 2: Ablation Study

Approach	EM	SemEqv
w/o Location	36 (23.8%)	51 (33.8%)
w/o Optimization Strategy	15 (9.9%)	41 (27.2%)
SemOpt	42 (27.8%)	64 (42.4%)

21 times of the success optimizations of the Direct baseline, and 1.65 to 2 times of the RAG baseline. For the Semantic Equivalence metric, SemOpt has 3.38 to 7.11 times of the Direct baseline, and 1.38 to 2 times of the RAG baseline. These results demonstrate the effectiveness of SemOpt in addressing code optimization tasks.

4.2.4 Complementarity of SemOpt and RAG. As shown in the SemOpt + RAG row of Table 1, combining the optimization results of both methods increases the number of successful optimizations by 17.19% to 33.33% compared to using SemOpt alone. This demonstrates the strong complementarity between SemOpt and RAG.

4.2.5 Practical Applicability of SemOpt. To evaluate the practicality of SemOpt in real-world scenarios, it is necessary to measure the proportion of proposed optimizations that are applicable. An optimization is considered valuable if it preserves the original semantics and achieves a certain degree of performance improvement (e.g., execution speed or resource efficiency), regardless of whether it matches the optimization result in the original submission.

Due to the substantial workload, we conducted a manual evaluation only on all optimization suggestions generated by SemOpt when using DeepSeek-V3. Among these, 89.86% of the optimizations were deemed to meet the criteria, demonstrating the high practical applicability of SemOpt in real-world scenarios.

Answer to RQ1: SemOpt achieves a significant improvement in producing correct optimization outcomes, reaching 37.5% to 27x more successful optimizations than all baselines under different LLMs. Furthermore, SemOpt and RAG exhibit strong complementarity. Besides, SemOpt demonstrates high practical applicability, with 89.86% of its generated optimizations preserving semantics while improving performance.

4.3 RQ2: Ablation Study

To verify the effectiveness of the design of SemOpt, we conduct an ablation study on the two functionalities associated with Semgrep rules by removing information in the prompt:

- **w/o Location:** Remove the location information from the LLM input prompt and provide only the optimization strategy. The LLM has to determine how to apply the optimization strategy to the whole input function.
- **w/o Optimization Strategy:** Remove the description of the optimization strategy from the LLM input prompt and provide only the location information. The LLM has to determine which strategy to use to optimize the given code segment.

```
for (auto dim : llvm::enumerate(op.broadcast_dimensions())) {
    output_to_input_dim[dim.value().getSExtValue()] = dim.index();
}
```

(a) The code segment requiring optimization.

```
@@ -227,1 +227,1 @@
- for (auto dim : llvm::enumerate(op.broadcast_dimensions()))
+ for (const auto& dim : llvm::enumerate(op.broadcast_dimensions()))
```

(b) The correct optimization result.

```
@@ -91,1 +91,1 @@
- for (auto it : llvm::enumerate(dimension_numbers.getOffsetDims()))
+ for (const auto& it : llvm::enumerate(dimension_numbers.getOffsetDims()))

@@ -103,1 +103,1 @@
- for (auto it : llvm::enumerate(result_ty.getShape()))
+ for (const auto& it : llvm::enumerate(result_ty.getShape()))
```

(c) The first code modification reference used by RAG.

```
@@ -103,1 +103,1 @@
- for (auto it : llvm::enumerate(is_outer_dim))
+ for (const auto &it : llvm::enumerate(is_outer_dim))
```

(d) The second code modification reference used by RAG.**Figure 5: Code optimization example for RQ3.**

We conducted experiments on 151 optimization tasks using DeepSeek-V3. The experimental results are shown in Table 2. Removing the location information led to a performance drop of 14.29%–20.31%, while removing the optimization strategy resulted in a decrease of 35.94%–64.29%. These findings demonstrate that both types of information provided by Semgrep rules contribute to the overall performance.

Answer to RQ2: In the Semgrep rules generated by SemOpt, both location and optimization strategy information significantly contribute to the overall performance.

4.4 RQ3: Generalization Across Codebases

We observed that, for some tasks where RAG achieved successful optimizations, it referenced other commits from the same codebase. In certain cases, the optimization strategies in the referenced commits were highly consistent with those required by the current task, thus providing substantial assistance.

An example is presented in Figure 5. Figure 5a shows the code snippet to be optimized, while Figure 5b illustrates the correct optimization approach, in which the loop variable in the for statement is modified with both const and reference qualifiers. Figures 5c and 5d present the top-1 and top-3 most similar examples retrieved by the RAG method as references for code optimization. All three code optimization commits originate from the TensorFlow [44] codebase and were submitted by the same developer on the same day, with largely consistent optimization strategies.

To further investigate the dependence of SemOpt and RAG on information from the same codebase, we implemented two modes for each method: Standard Mode and Degraded Mode.

- **Standard Mode:** Correspond to the standard implementation used in RQ1. In this mode, we exclude the exact match of the

Table 3: The performance comparison of SemOpt and prompt engineering techniques on different LLMs.

Approach	Mode	DeepSeek-V3		GPT-4.1		Gemini-2.5-Pro (on 40 problems)	
		EM	SemEqv	EM	SemEqv	EM	SemEqv
RAG	Standard	25 (16.6%)	36 (23.8%)	17 (11.3%)	32 (21.2%)	6 (15.0%)	9 (22.5%)
	Degraded	6 (4.0%)	10 (6.6%)	5 (3.3%)	15 (9.9%)	1 (2.5%)	5 (12.5%)
SemOpt	Standard	42 (27.8%)	64 (42.4%)	28 (18.5%)	44 (29.1%)	12 (30.0%)	18 (45.0%)
	Degraded	42 (27.8%)	63 (41.7%)	23 (15.2%)	42 (27.8%)	9 (22.5%)	16 (40.0%)

Table 4: In-the-wild evaluation result

Repo	Perf Improvement ↑		# Test Cases	
	Max	Avg	↑ ≥ 5%	↑ ≥ 10%
RocksDB	5.04%	2.41%	1/2	0/2
Redis	6.14%	1.68%	3/21	0/21
gRPC	35.48%	3.10%	2/16	1/16
LevelDB	218.07%	10.40%	2/22	1/22
spdlog	20.00%	3.27%	6/45	4/45

commit or code from the strategy library and the RAG knowledge base to avoid data leakage, but there may still be similar commits in the same codebase, such as the example in Figure 5.

- **Degraded Mode:** Corresponds to a reduced implementation that excludes the knowledge in the same codebase. During optimization, all entries in the knowledge base corresponding to historical commits from the same codebase are excluded, while all other entries remain unchanged.

As shown in Table 3, when switching from Standard Mode to Degraded Mode, the number of successful optimizations achieved by SemOpt decreases by only 0% to 25%, while RAG experiences a substantially larger reduction of 44.44% to 83.33%. Under both metrics in Degraded Mode, SemOpt achieves 2.8 to 9 times as many successful optimizations as RAG. This demonstrates that the majority of RAG’s successful optimizations depend on historical commits from the same codebase, whereas SemOpt shows significantly less reliance, indicating that SemOpt has stronger generalization capability across different codebases.

Answer to RQ3: RAG demonstrates a significantly higher dependence on information from other commits within the same codebase compared to SemOpt, indicating that SemOpt has stronger generalization capabilities across codebases.

5 In-the-Wild Evaluation

In this section, we evaluate the real-world practicality of SemOpt by identifying and optimizing hotspot functions in five large-scale C/C++ projects, then measuring performance improvements using their comprehensive performance test suite.

5.1 Experimental Setup

5.1.1 Benchmark. We collected five large-scale C/C++ projects: RocksDB [9], Redis [39], gRPC [18], LevelDB [19], and spdlog [30]. These projects are widely used in the industry and span a variety of domains, including storage engines, in-memory databases, distributed communication, remote procedure calls, and logging systems. Additionally, each of these projects provides a comprehensive performance test suite, which can be used to quantitatively measure the degree of optimization.

For each project, we cloned the latest available version and wrote a script to compile the code and run the built-in unit and performance tests. The performance testing results of these 5 projects contain between 2 and 45 distinct test cases, each test case typically reflecting the performance of a different functionality within the project. We designed a consistent method for extracting the relevant data from each project.

5.1.2 Metrics. For each project, we attempt to apply the optimization, compile the code, and then execute the unit tests on the optimized code. If the tests are passed, we conduct the performance tests 6 times each on both the original and optimized versions of the project. To avoid issues such as cold starts, we discard the results from the first run and report the average performance improvement in the remaining 5 runs for each test case.

Depending on the implementation of the test case, the performance may be reported as a number where higher values indicate better performance (e.g., processed items per second) or lower values indicate better performance (e.g., execution time). We unified the performance improvement in both types as the improvement of speed (i.e., 100% improvement if the number increases from 10 to 20 in the former type, or decreases from 20 to 10 in the latter type). Formally, let x and y denote the performance results for a given test case before and after optimization. The improvement ratio is calculated as $\frac{y-x}{x}$ for the former type or $\frac{x-y}{y}$ for the latter type.

To evaluate the effectiveness of the optimization, we employed the following two metrics.

- **Perf Improvement ↑:** The maximum and average improvement ratio of all test cases.
- **# Test Cases where ↑ ≥ n%:** The number of test cases among all test cases where the improvement ratio exceeds n%.

5.2 Optimization Process

For each project, we first identify hotspot functions through profiling, then generate potential optimizations for each function, and finally combine optimizations across all hotspot functions.

5.2.1 Identify Hotspot Functions. Since a performance test typically only covers a small set of functions that need to be optimized, we first identify the hotspot functions for each test case. We used perf [35] and gcov [11] to profile the execution of the performance test. Based on the profiling result, we then identified all functions whose execution time accounted for more than 0.1% of the total runtime as hotspot functions in the performance tests. Only these functions are targets for subsequent optimization.

5.2.2 Generate and Combine Optimization Results. For each project, we first apply SemOpt to optimize all identified hotspot functions.

Next, we evaluate all optimization results according to the following criteria: an optimization is considered effective if it achieves a performance improvement exceeding 5% on at least one test case while causing no more than 2% performance degradation on any remaining test cases. This approach ensures that selected optimizations provide substantial performance benefits without introducing significant regressions elsewhere.

We then automatically compose the optimization results across different functions through the following process. For each function, if multiple optimization variants satisfy our effectiveness criteria mentioned before, we select the variant with the highest total improvement score – computed by adding the performance improvement percentages across all test cases. If no variants meet the criteria, the original implementation is preserved for that function. Finally, we assemble the selected optimizations for all hotspot functions to generate the final optimized version of each project.

5.2.3 Implementation Details. Unlike the empirical evaluation in Section 4, this experiment relies on the performance testing results. Therefore, it is crucial to maintain steady performance measurement throughout the experiment. The experiments were conducted on a Linux server equipped with four Intel Xeon Platinum 8270 CPUs (104 cores @ 2.70 GHz) and 256 GB of RAM. To minimize the impact of fluctuations on the performance testing results, we ensured that no other resource-intensive programs were running on the server when executing the performance tests.

5.3 Optimization Result

As shown in Table 4, the maximum improvement on each project ranges from 5.04% to 218.07%, with an average improvement between 1.68% and 10.40%. Each project contains 2 to 45 test cases, among which 1 to 6 test cases exhibit improvements greater than 5%, and 0 to 4 test cases show improvements exceeding 10%.

The authors further manually validated the optimized code for semantic correctness. All of them were semantically equivalent to the original program.

Finding: We demonstrate that SemOpt can effectively optimize five large-scale C/C++ projects in the real world while maintaining correctness.

6 Discussion

6.1 Significance of Technological Contributions

SemOpt introduces two components as its main technical contributions: clustering optimization strategies, and generating corresponding Semgrep rules for each strategy. Omitting either component would result in a substantial increase in optimization overhead and a decline in optimization effectiveness:

- **Clustering optimization strategies:** In the clustering results, we remove all noise points, and only 22% of the commits are ultimately used to generate Semgrep rules. If this step were omitted, the overall optimization overhead would increase by 5x, and a substantial number of ineffective optimization strategies might be introduced, negatively impacting the final performance of SemOpt.
- **Generating Semgrep rules:** Without utilizing Semgrep, given a piece of code requiring optimization, it would first be necessary to partition the code into multiple chunks and then use an LLM to evaluate, one by one, whether each strategy in the strategy library is applicable. In our benchmark with 151 code optimization tasks, assuming each input code can be divided into 5 chunks and the strategy library contains approximately 150 optimization strategies, this would require $151 * 5 * 150 = 113,250$ LLM invocations in total. Such additional computational overhead is too costly for practical use.

6.2 Limitation

SemOpt focuses exclusively on optimizing code within individual functions. Future work may extend SemOpt to support code optimization at a broader scope, such as multi-function modifications.

Additionally, the types of optimization strategies we can currently implement are constrained by the expressiveness of the Semgrep static analyzer. Incorporating a better analysis tool may extend the capability of SemOpt.

6.3 Threats to Validity

The main internal threat comes from potential data leakage. We already excluded exact matches in the strategy library, but there is still a possibility of data leakage through LLMs, since some experiment subjects may be included in their training set. However, as demonstrated in Section 4, directly prompting LLMs to optimize code yields suboptimal results. SemOpt outperforms other baselines with the same LLM (hence under the same degree of data leakage). Furthermore, Section 5 shows that SemOpt can generate new optimization suggestions for multiple projects in the wild, which are unlikely to be memorized by the LLM. These findings indicate that this threat has a limited impact on our findings.

One external threat comes from the selection of LLMs and the benchmark. For LLMs, we evaluated SemOpt on three popular LLMs from different vendors, all of which demonstrated that SemOpt achieves superior optimization performance over baseline methods. Moreover, our implementation of SemOpt is model-agnostic, and we provide the replication package so that other researchers can experiment on arbitrary LLMs. For the benchmark, we used the 100 most starred C/C++ databases on GitHub and randomly selected eligible commits from these databases to serve as our benchmarks.

Given the representativeness and diversity of the benchmark problems, we believe the external threat to the benchmark is limited. Furthermore, our approach has successfully optimized large-scale projects in the wild, further confirming the effectiveness of our approach.

7 Related Work

Early code optimization research mainly used rule-based and analytical methods targeting specific inefficiencies [4, 7, 17, 24, 29]. These approaches rely on expert-defined rules, which are labor-intensive and difficult to scale. In contrast, SemOpt automatically applies diverse optimization strategies and offers strong scalability.

With the popularity of deep learning, researchers have explored to train dedicated performance-improving models, such as VQ-VAE [1], Supersonic [2], and DeepDev-PERF [15]. However, these dedicated models have been shown to have lower performance than LLM-based approaches [16], probably because the knowledge obtained from large-scale pre-training is often critical for performing the optimization.

Recently, researchers also explored LLM-based approaches for code optimization. Shypula et al. [43] builds a benchmark for code optimization and evaluates the performance of various prompting techniques on this benchmark. A key finding of this study is that RAG could significantly outperform direct prompting. Our work confirms this finding and further proposes a new approach that significantly outperforms RAG. RAPGen [16] performs API replacement-based code optimization for C#. Compared to SemOpt, RAPGen is limited to a single optimization strategy, whereas SemOpt supports diverse strategies. SBLLM [13] combines multiple optimized versions of the same code segment for better overall optimization. EffiCoder [21] identifies performance bottlenecks by executing test cases and utilizes the obtained profiling data to guide LLMs in optimizing inefficient code segments, while PerfCodeGen [36] employs bottleneck test cases as iterative feedback signals to prompt LLMs for progressive code refinement. Compared to our work, the latter three studies all explore orthogonal aspects and could potentially be combined with SemOpt in the future.

8 Conclusion

In this paper, we propose SemOpt, an LLM-based automatic code optimization approach that integrates static program analysis techniques. SemOpt consists of three components: an optimization strategy library builder, an automatic static program analysis rule generator, and a library-based optimizer. Extensive experiments conducted on three popular LLMs demonstrate that SemOpt can effectively localize and retrieve appropriate optimization strategies on large projects, thereby guiding the LLMs to perform code optimizations. Furthermore, our in-the-wild evaluation shows that SemOpt significantly improves the performance of real-world projects.

Data Availability

Artifacts of this paper, including the implementation of SemOpt, all baseline implementations, and evaluation results, are available at <https://figshare.com/s/0391db49a3620ed53a1f>.

References

- [1] Binghong Chen, Daniel Tarlow, Kevin Swersky, Martin Maas, Pablo Heiber, Ashish Naik, Milad Hashemi, and Parthasarathy Ranganathan. 2022. Learning to improve code efficiency. *arXiv preprint arXiv:2208.05297* (2022).
- [2] Zimin Chen, Sen Fang, and Martin Monperrus. 2024. Supersonic: Learning to generate source code optimizations in C/C++. *IEEE Transactions on Software Engineering* (2024).
- [3] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875* (2022).
- [4] John Cocke. 1970. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*. 20–24.
- [5] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. Perscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.
- [6] Google DeepMind. 2025. Gemini 2.5: Our most intelligent AI model. *Google AI Blog* (March 2025). <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/> GA release, Deep Think enhancement, multimodal capabilities; accessed July 2025.
- [7] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2015. Performance problems you can fix: A dynamic analysis of memoization opportunities. *ACM SIGPLAN Notices* 50, 10 (2015), 607–622.
- [8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.
- [9] Facebook. 2025. RocksDB. <https://github.com/facebook/rocksdb>.
- [10] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.
- [11] Free Software Foundation. 2025. gcov—a Test Coverage Program. <https://gcc.gnu.org/onlinedocs/gcov/Gcov.html>.
- [12] Internationale Organisation für Normung. 2011. *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuARE)—System and Software Quality Models: Ingénierie Des Systèmes Et Du Logiciel—Exigences Des Qualité Et Évaluation Des Systèmes Et Du Logiciel (SQuARE)—Modèles de Qualité Du Système Et Du Logiciel*. ISO.
- [13] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. *arXiv preprint arXiv:2408.12159* (2024).
- [14] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.
- [15] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. 2022. Deepdev-perf: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 948–958.
- [16] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2023. Rapgen: An approach for fixing code inefficiencies in zero-shot, 2024. URL <https://arxiv.org/abs/2306.17077> (2023).
- [17] Rafail Giavrimis, Alexis Butler, Constantin Cezar Petrescu, Michail Basios, and Santanu Kumar Dash. 2021. Genetic optimisation of C++ applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1180–1182.
- [18] Google. 2025. gRPC. <https://github.com/grpc/grpc>.
- [19] Google. 2025. LevelDB. <https://github.com/google/leveldb>.
- [20] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
- [21] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. 2024. SWIFTCODER: Enhancing Code Generation in Large Language Models through Efficiency-Aware Fine-tuning. *arXiv preprint arXiv:2410.10209* (2024).
- [22] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2023. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research* 24, 251 (2023), 1–43.
- [23] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 155–170.
- [24] Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2020. Cadet: Debugging and fixing misconfigurations using

- counterfactual reasoning. *arXiv preprint arXiv:2010.06061* (2020).
- [25] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
 - [26] Weichen Li, Albert Jan, Baishakhi Ray, Chengzhi Mao, Junfeng Yang, and Kexin Pei. 2025. EditLord: Learning Code Transformation Rules for Code Editing. *arXiv preprint arXiv:2504.15284* (2025).
 - [27] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Cheng-gang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
 - [28] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).
 - [29] William M McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (1965), 443–444.
 - [30] Gabi Melman. 2025. spdlog. <https://github.com/gabime/spdlog>.
 - [31] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.
 - [32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
 - [33] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 237–246.
 - [34] OpenAI. 2025. GPT-4.1 Model Release. <https://openai.com/index/gpt-4-1/>. Includes GPT-4.1, GPT-4.1-mini, and GPT-4.1-nano; context window up to 1 million tokens; knowledge cutoff June 2024.
 - [35] Linux Kernel Organization. 2025. perf: Linux profiling with performance counters. <https://perf.wiki.kernel.org/>.
 - [36] Yun Peng, Akhilesh Deepak Gotmare, Michael R Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2025. Perfcoder: Improving performance of llm generated code with execution feedback. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 1–13.
 - [37] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative type inference for python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 988–999.
 - [38] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).
 - [39] Redis. 2025. Redis. <https://github.com/redis/redis>.
 - [40] Nils Reimers and Iryna Gurevych. 2021. all-MiniLM-L6-v2. Available at <https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>. Accessed: 2025-07-18.
 - [41] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. *Okapi at TREC-3*. British Library Research and Development Department.
 - [42] Inc. Semgrep. 2020–2025. Semgrep. <https://github.com/semgrep/semgrep>. Accessed: 2025-07-16.
 - [43] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867* (2023).
 - [44] tensorflow. 2025. TensorFlow. <https://github.com/tensorflow/tensorflow>.
 - [45] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems* 33 (2020), 5776–5788.
 - [46] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).
 - [47] He Ye and Martin Monperrus. 2024. Iter: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.
 - [48] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087* (2023).