POLO: An LLM-Powered Project-Level Code Performance Optimization Framework

Jiameng Bai[1], **Ruoyi Xu**[1], **Sai Wu**[1,2*], **Dingyu Yang**[3], **Junbo Zhao**[1] and **Gang Chen**[1]
[1]College of Computer Science and Technology, Zhejiang University
[2]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
[3]The State Key Laboratory of Blockchain and Data Security, Zhejiang University
{baijiameng, noyexception, wusai, yangdingyu, j.zhao, cg}@zju.edu.cn

Abstract

Program performance optimization is essential for achieving high execution efficiency, yet it remains a challenging task that requires expertise in both software and hardware. Large Language Models (LLMs), trained on high-quality code from platforms like GitHub and other open-source sources, have shown promise in generating optimized code for simple snippets. However, current LLM-based solutions often fall short when tackling project-level programs due to the complexity of call graphs and the intricate interactions among functions. In this paper, we emulate the process a human expert might follow when optimizing project-level programs and introduce a three-phase framework **POLO** (PrOject-Level Optimizer) to address this limitation. First, we profile the program to identify performance bottlenecks using an iterative weighting algorithm. Next, we conduct structural analysis by scanning the project and generating a graph that represents the program's structure. Finally, two LLM agents collaborate in iterative cycles to rewrite and optimize the code at these hotspots, gradually improving performance. We conduct experiments on open-source and proprietary projects. The results demonstrate that POLO accurately identifies performance bottlenecks and successfully applies optimizations. Under the `O3` compilation flag, the optimized programs achieved speedups ranging from 1.34x to 21.5x.

1 Introduction

Program performance issues, often referred to as performance bugs, are defects that arise from inefficient programming practices, such as the use of suboptimal data structures. These defects lead to excessive processing overhead and unnecessary resource allocation. Even worse, performance bugs typically do not cause program malfunctions, making them difficult to detect and particularly elusive [Nistor *et al.*, 2013]. While tools like FlameGraph [Gregg, 2016] and Valgrind [Nethercote and Seward, 2007] are available for performance analysis, fixing performance bugs requires more advanced expertise than simply writing the code, making it a highly challenging task. Sometimes, performance issues may arise from workload variability or deployment environment. In this paper, we focus on performance problems caused by inefficient implementations at the source code level. Bottlenecks due to other factors are beyond the scope of this paper.

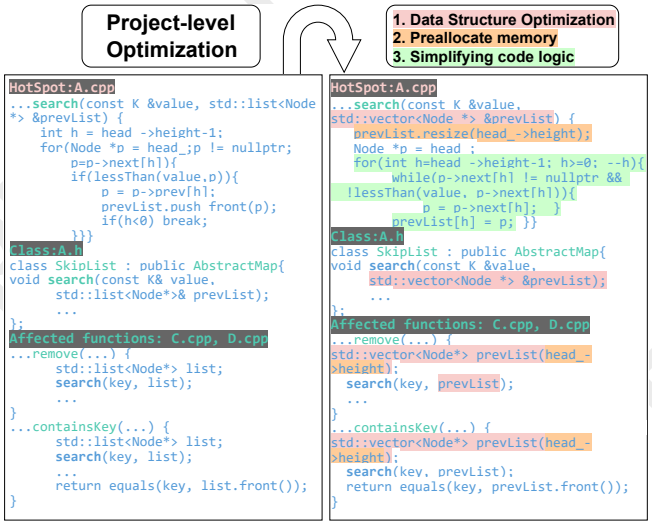

Figure 1: An example of project-level program optimization. All modules affected by changes to `search` must be adapted.

Traditional methods typically focus on specific types of defects, such as inefficient loops [Song and Lu, 2017; Xiao *et al.*, 2013], redundant computations [Della Toffola *et al.*, 2015], etc. With recent advances in generative language models, researchers [Chen *et al.*, 2024; Shypula *et al.*, 2023] have started collecting pre- and post-optimization code to fine-tune LLMs for addressing a wider range of performance issues. However, these efforts generally focus on optimizing individual functions (e.g., LeetCode problems) or even specific lines, making them insufficient to meet the project-level optimization demands in real-world scenarios.

Recently, some works have shifted their focus to function-level optimization. For example, DeepPerf [Garg *et al.*, 2022] collects code snippets of the same function before and after optimization in C# projects to fine-tune code models. Its follow-up, RAPGen [Garg *et al.*, 2023], retrieves knowledge

*Corresponding author.

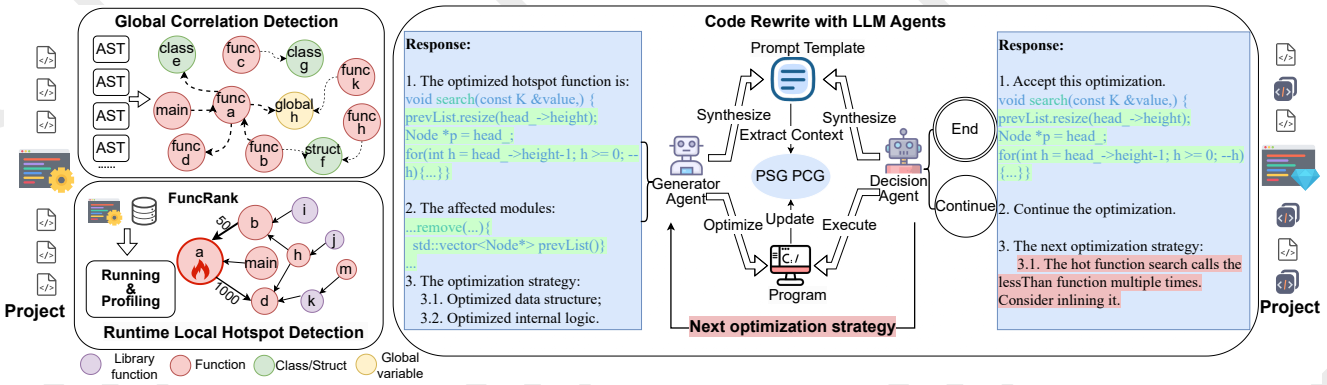

Figure 2: Overview of the POLO framework.

from pre-built databases and leverages large models to perform optimizations based on historical examples. However, RAPGen's reliance on code matching limits it to handling additions, deletions, and modifications related to the misuse of common C# API. This approach requires a knowledge base constructed from before-and-after optimization changes in code snippets of specific functions, which incurs significant costs for collection and maintenance. Besides, it is challenging to adapt it to other types of bugs and programming languages.

In the absence of a dedicated knowledge base for the project, we must rely on common sense and practical experience for code optimization. Fortunately, LLMs, having been trained on vast repositories of projects written in various languages, possess a general understanding of what well-optimized code looks like. If we can effectively guide LLMs during program optimization, we can establish a zero-shot method for performance debugging. As illustrated in Fig. 1, we begin by reviewing how a human expert approaches project-level code optimizations and identify the challenges:

- Profiling and Hotspot Identification: In Fig. 1, the optimization is centered around the hotspot **`search`**. The key to effective optimization is profiling the program to identify the hotspot that most impacts performance. It is crucial as execution paths and bottlenecks vary across programs. In complex programs, the call graph can be intricate, making it challenging to pinpoint the hotspot.
- Project-Level Correlation Detection: For project-level performance optimization, it is essential to understand the context of a hotspot function, including its position in the program and relationships with other modules. In Fig. 1, the hotspot **`search`** calls the **`lessThan`**. It is invoked by **`containsKey`** and **`remove`**, and is a member of the **`SkipList`** class. These insights provide a comprehensive understanding of the program's structure, enabling more effective optimization by the expert.
- Code Rewrite: After identifying the chain of bottlenecks, we attempt to rewrite the code to improve performance. However, this is a non-trivial task. The program shown in Fig. 1 can be optimized in various aspects, including data structures, memory allocation, and code logic. The diverse nature of performance issues across different projects requires a human expert to possess a broad range of skills and knowledge across multiple domains. This is the most challenging aspect of optimization, as it demands expertise in multiple areas to effectively address and resolve these varied performance issues.

In this paper, we propose the **POLO** (PrOject-Level Optimizer) to tackle the challenge of project-level code optimization by simulating the optimization process of a human expert. POLO is organized into the following three modules:

Runtime Local Hotspot Detection: Program hotspots can only be identified after actual execution, necessitating the use of runtime analysis to detect local hotspots, as discussed in Sec. 3.1. By running the program under real workloads and using instrumentation tools, POLO can capture the program's runtime behavior, ensuring the detection of genuine performance bottlenecks. To model the program's runtime states, we employ a meticulously designed finite state machine to automatically parse profiling results and construct a directed graph, the Program Call Graph (PCG), in which nodes represent functions and edges denote function call relationships. To precisely identify hotspot, we apply a PageRank-like method, Func_Rank, on the PCG to rank the functions, considering both their execution time and impact within the call stack.

Global Correlation Detection: Runtime analysis alone cannot reveal the complete structure of the program. For example, if there are numerous "if-else" branches and some are not executed during runtime, critical information will be missing. A comprehensive understanding of the program's structure is essential for grasping its functionality and semantics. As discussed in Sec. 3.2, we perform a static analysis of the entire project, constructing a Program Structure Graph (PSG) that represents the structural information of the program, where nodes denote key elements and edges represent their relationships. The nodes and edges sufficiently encapsulate program information to facilitate subsequent optimization.

Code Rewrite with LLM Agents: LLMs, as programming experts, have already achieved significant success in code generation [Zhu *et al.*, 2024; Zheng *et al.*, 2023b; Zheng *et al.*, 2023a; Ni *et al.*, 2023]. We leverage multiple agents in an interactive and iterative manner to perform optimization. The Generator Agent extracts neighboring nodes of the hotspot from PSG and PCG, crafts prompts, and inter-

	Elements (Nodes)		Relations (Edges)	
	Type	Attribute	Type	Attribute
Correlation analysis	Class/Struct	name, body, template, file	Static refer	referer, referee
	Function	name, parameter, return type, body, template, file	Inheritance	superclassOf, subclassOf
	Global/Static variable	name, value, file	Ownership	hasmember, ismember
Runtime analysis	Function	custom, library, time, file	Dynamic call	caller, callee, frequency

Table 1: All elements and relations extracted from global correlation analysis and local runtime analysis.

acts with the LLM to generate optimized code. The Decision Agent executes the modified program, evaluates performance changes, and decides on further optimization or improvement directions. This iterative loop ensures optimizations are guided by real-time performance impact for targeted improvements.

Overall, the contributions of this paper are as follows:

- To the best of our knowledge, POLO is the first end-to-end project-level program optimization framework, encompassing global program analysis, local hotspot detection, and the code rewriting capabilities of large models, enabling us to make a exploration into the relatively underexplored field of performance optimization.
- We propose using the Program Structure Graph (PSG) to model project architecture, incorporating richly attributed nodes and edges to facilitate a comprehensive understanding of the program's semantic context. We advocate for an automated system to manage information during the runtime, constructing a Program Call Graph (PCG). By applying a PageRank-like algorithm to this graph, our profiling process can effectively identify hotspot.
- We propose a collaborative, iterative method involving two LLM agents to optimize programs based on hotspots and program context. In each optimization cycle, the agent continuously senses performance changes from the environment, deciding whether to accept the current optimization and guiding subsequent optimizations.
- We conduct experiments on six open-source and private projects. The results indicate that POLO successfully identifies and optimizes hotspot, achieving speedups ranging from 1.34x to 21.5x under the `O3` compilation flag.

2 Related Work

2.1 LLMs for Code Generation

Generative language models have achieved remarkable accomplishments [Yu *et al.*, 2024; Yang *et al.*, 2023; Du *et al.*, 2024; Huang *et al.*, 2023; Liu *et al.*, 2024b; Liu *et al.*, 2024a; Wang *et al.*, 2024; Zan *et al.*, 2022] in the automated code generation field. They are capable of generating code based on user instructions or the context of the program. Currently, there are many generative models specifically tailored for code generation, like CodeBert [Feng *et al.*, 2020] and CodeLlama [Roziere *et al.*, 2023]. The large language models (LLMs), represented by GPT 4 [Achiam *et al.*, 2023], have further propelled research to its peak. LLMs have been proven to effectively understand and generate code. They have achieved notable results on code benchmarks. However, it is worth noting that these benchmarks focus on the ability to generate

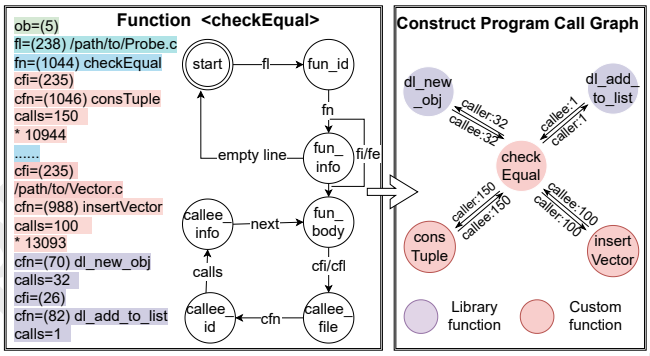

Figure 3: A finite state machine that designed for the profiling tool.

a single code file. Considering that real-world codes typically exist at the project-level, initiatives like CodePlan [Bairi *et al.*, 2024] have begun to leverage LLMs for project-level code migration tasks. RepoAgent [Luo *et al.*, 2024] explores the documentation generation task. Nevertheless, such tasks remain relatively unexplored.

2.2 Performance Optimization

Performance optimization, also referred to performance bug repair, involves correcting code that causes poor performance due to inefficient data structures or unnecessary operations. Early methods aim to address specific types of performance deficiencies, such as inefficient loops [Song and Lu, 2017; Xiao *et al.*, 2013], suboptimal data structures [Xu *et al.*, 2010], and redundant computations [Della Toffola *et al.*, 2015]. Generative models have brought new directions for performance optimization. PIE4Perf [Shypula *et al.*, 2023] and Supersonic [Chen *et al.*, 2024] collect pre- and post-optimization versions of numerous C++ programming solutions and fine-tune existing generative models to perform optimization. SBLLM [Gao *et al.*, 2024] iterates the optimization process from the perspective of search. However, their optimizations are mainly within single files (On the algorithm competition platform, e.g., AtCoder and Codeforces), which cannot meet the requirements of project-level performance repair in real-world scenarios. Recently, DeepPERF [Garg *et al.*, 2022] and Rapgen [Garg *et al.*, 2023] have begun attempting program optimizations for C# language. DeepPERF collects pre- and post-optimization code pairs and fine-tunes the code language models. Similarly, RAPGen utilizes the same code pairs to build an optimization knowledge base to generate prompts for LLMs to perform optimizations. However, it focuses primarily on fixing misuse of common C# API and lacks the capability to optimize other types of bugs. Overall, project-level optimization remains a relatively unexplored area.

3 Method

The architecture of POLO is shown in Fig. 2, consisting of **Runtime Local Hotspot Detection**, **Global Correlation Detection**, and **Code Rewrite with LLM Agents**.

3.1 Runtime Local Hotspot Detection

We use the Callgrind profiling tool [Weidendorfer, 2012] for dynamic analysis. Unfortunately, the profiling results generated by Callgrind are not human readable. To enable further analysis of runtime critical information, we thoroughly analyze Callgrind's output and develop a Finite State Machine (FSM) for automated parsing. As shown in Fig. 3, the left side displays the highly simplified version of Callgrind output. We employ the FSM on the right side to parse each file, function, and call information involved at runtime line by line. Despite the highly varied output of Callgrind due to its built-in compression methods, our FSM can extract the execution cost and call relationships for each function, including those from numerous standard and third-party libraries.

Based on the parsed call information, we define the directed graph, referred to as the Program Call Graph (PCG), to represent the intricate call relationships within the program. Formally, $PCG = (V^c, E^c)$, where V^c is the set of functions executed during the program's running (including library functions), and E^c represents the dependencies between these functions. Both nodes and edges contain rich attributes. For example, the node `checkEqual` in Fig. 3 is characterized by $A_i^c = \{name : \text{“}checkEqual\text{”}, type : \text{“}custom\text{”}, time : 25\%\}$, which indicates that $i \in V^c$ is a custom function, and its execution time contributes 25% of the total cost (excluding its calling relationships). The $A_{ij}^c = \{type : \text{“}caller\text{”}, count : 100, time : 20\%\}$ describes an edge $(i, j) \in E^c$, where function i (the caller) invokes function j (the callee) 100 times, and the call from i to j contributes 20% of the total execution cost.

Considering cases where time-consuming function is caused by errors in their callers/callees or where non-optimizable library functions become performance bottlenecks, it is evident that relying solely on function execution time is insufficient for hotspot identification. A thorough analysis of the lengthy call stack and execution duration is necessary to pinpoint hotspots effectively. To address this, we propose a PageRank-like ranking method: **Func_Rank**. We draw inspiration from the PageRank [Page *et al.*, 1999] algorithm. Func_Rank simultaneously considers both the time-consuming nature of functions and their influence within the PCG. Specifically, we analogize functions to web pages, with function calls viewed as links between them, and the execution cost of the function representing its inherent influence. The called functions propagate time cost to their callers. Formally, we first compute the weights for edges of the callee type. The weight is shown as:

$$w_{ji} = \frac{A_{ji}^c[time]}{\sum_{u \in caller(j)} A_{ju}^c[time]} \quad \text{s.t.} \quad A_{ji}^c[type] == \text{“}callee\text{”} \tag{1}$$

where A_{ju}^c is the set of edges from node j to node u, and $A_{ju}^c[time]$ denotes the time cost for this invocation. Here, $caller(j)$ represents the set of nodes that call j.

We use I_i to represent the final score of the function i, which is derived from the function's real execution cost and its influence within the call graph. This value is used to rank and identify hotspot functions.

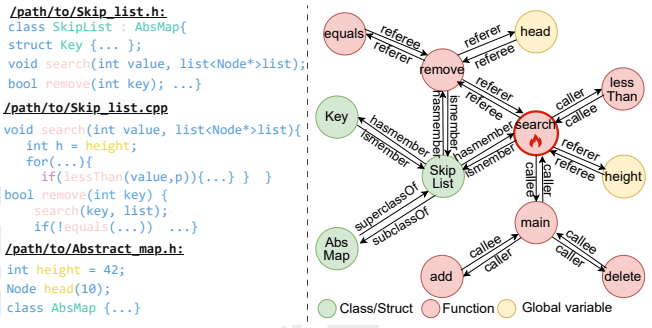

Figure 4: An example of program structure graph.

$$I_i = (1 - \alpha)A_i^c[time] + \alpha(\sum_{j \in callee(i)} w_{ji} I_j) \tag{2}$$

where j is a callee of node i. α is the dumping factor, which balances the functions' execution time and the influence propagated from other nodes, with a default value of 0.5. We apply the above equation iteratively until the values converge.

3.2 Global Correlation Detection

Conducting a global static analysis of the project is necessary. It not only aids in understanding the context of functions but also ensures proper adaptation of functions in the call stack impacted by optimizations involving changes to function signatures. Languages like C++ are among the most challenging due to their support for multiple programming paradigms, template and other advanced features. Its status as the primary choice for performance-critical algorithm motivates us to select it as the example for program analysis in this work.

We use Clang's LibTooling [Team, 2007] to construct an Abstract Syntax Tree (AST) for each source file. The nodes of the ASTs are analyzed through recursive traversal, extracting both node attributes and relationships. The analysis process is detailed in the Algorithm 1 in Appendix A. As shown in Table 1, we extract Classes, Functions, Global variables, and their attributes. Class is used to define a custom data type. Function, being the basic building block of a program, is a key targets for optimization. Global variable is used to store essential information shared across various modules.

We define a directed graph, referred to as the Program Structure Graph (PSG), to represent the structural relationships within the program, as illustrated in Fig. 4. Formally, the graph is denoted as $PSG = (V^s, E^s)$, where V^s represents the set of vertices, each corresponding to a function, class, or global variable identified through program analysis. The edges, E^s, capture the relationships between these vertices. Each vertex $u \in V^s$ is associated with a set of attributes A_u^s, which describe its properties. For instance, $A_u^s = \{type : \text{“}function\text{”}, name : \text{“}Search\text{”}, ...\}$ indicates that the vertex u represents a function named “Search”. To handle potential naming conflicts, each vertex also includes a namespace attribute, ensuring uniqueness among functions with identical names. Similarly, edges $(u, v) \in E^s$ are annotated with attributes A_{uv}^s, which describe the attribute of the

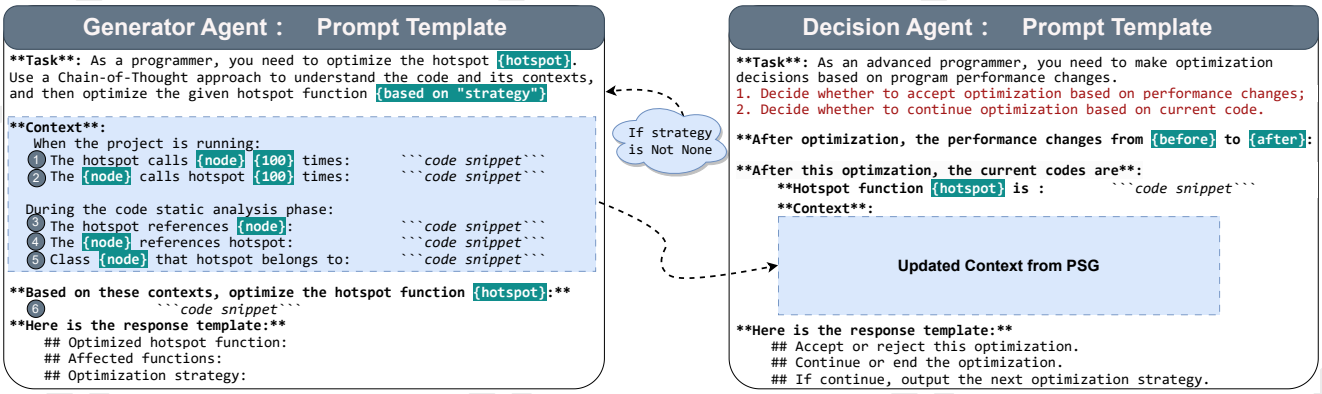

Figure 5: Prompt templates for the Generator Agent and Decision Agent.

relationship between vertices u and v. For instance, an edge $A^s_{uv} = \{type : \text{“}ismember\text{”}\}$ implies that there is an edge $(u, v) \in E^s$ between nodes u and v, where the relationship $u \to v$ indicating u is member of v; conversely, $v \to u$ is the “$hasmember$”, indicating v is parent of u. This information facilitates the extraction and assembly of each element in subsequent steps, such as constructing a function's signature from its template, name, return type, and parameter list.

3.3 Code Rewrite with LLM Agents

As shown in Fig. 2, we adopt two LLM agents for iterative optimization: **Generator Agent** and **Decision Agent**. Human programmers are sometimes limited by their own knowledge domains and may inadvertently write inefficient code. Leveraging Generator Agent as a code generator can mitigate these issues by unleashing their comprehensive and advanced programming capabilities. Decision Agent incorporates performance changes as feedback. It can effectively guide the LLM to optimize while avoiding negatively impact code.

Generator Agent

An important fact is that the **context of function** aids LLM in better understanding its functionality, thereby facilitating optimization. Additionally, the context helps the LLM comprehend the call stack of hotspot and their connections with other modules, enabling broader optimization rather than focusing solely on isolated code segments. Hence, we extract the neighboring nodes of the hotspot function as program context.

Prompt Synthesis. For the hotspot node u, its set of neighboring nodes and associated edges from PSG and PCG are denoted as $N^{node}(u) = \{v \in V^s \cup V^c \mid (u, v) \in E^s \cup E^c\}$ and $N^{edge}(u) = \{e = (u, v) \mid e \in E^s \cup E^c\}$. We obtain the neighboring nodes and their attributes as: $\{(v, A_v) \mid v \in N^{node}(u)\}$. Taking the `search` node in the Fig. 4 as an example, we can derive its context `lessThan`, `remove`, and `SkipList` from $N^{node}(u)$. Based on the edge attributes in $N^{edge}(u)$, we can derive that `search` has a call relationship with `lessThan` and is a member of `SkipList`. We synthesize neighboring nodes into different prompts based on the types of edges. The detailed template is shown in Fig. 5.

As shown in Fig. 5, the dynamic relationships are included in parts ① ② of the prompt, while the static relationships are covered in parts ③ ④ ⑤. Finally, the hotspot function itself ⑥ and the return format follow, completing the template synthesis. Since each node has attributes, we can concatenate the attributes (return type, name, parameters, body) to form the code snippet in the template.

Optimize with LLM. We send the assembled prompt to the LLM to optimize the hotspot. We intentionally inject the node ID of each node into the synthesized prompt and require the LLM's output to include the ID, which is given in the PSG. By parsing the response, we can automatically write the modified code back into the source files. It facilitates cross-file and cross-function modifications, making the optimization process process more seamless and efficient.

Decision Agent

The code generated by the Generator can impact performance both positively and negatively. Unnecessary optimizations may degrade performance. Therefore, tracking performance changes is crucial to retain only beneficial optimizations.

Program Execution and PSG Update. The program needs to be re-executed to assess the performance. We record both the pre- and post-optimization performances. Meanwhile, since the function bodies, and other modules may have changed, the PSG needs to be updated to reflect the newest version of program. This ensures that the PSG synchronizes the modified program, maintaining consistency for future steps.

Make Decision. As shown in Fig. 5, we track prompt changes and retrieve the current hotspot context based on the updated PSG. The LLM decides whether to accept or reject these changes, and whether to continue or terminate the optimization. If it continues, it provides additional strategies, which are passed to the Generator Agent for program optimization. This iterative process persists, with the Decision Agent evaluating each cycle, until further optimization is deemed unnecessary, at which point the process terminates.

4 Experiments

In real world, the compile level `O0` is used during development, and `O3` enables aggressive optimizations, aiming to produce highly optimized code for production needs. We consider both the `O0` and `O3` flags. The goal is to verify whether the

Level	Project	Language	File	Class	Functions	Lines	Type
Easy	Quant [Smidt, 2012]	C++	5	1	9	315	Quantitative Trade Framework
	AStar (Private)	C++	10	9	18	484	A* search Algorithm
Medium	SkipList (Private)	C++	6	7	44	886	SkipList Implementation
	AES [Conte, 2015]	C	3	0	31	1409	Crypto Algorithm
Hard	KGraph [DBAIWangGroup, 2021]	C++	8	22	109	3937	Nearest Neighbor Search
	MiniSQL (Private)	C++	129	130	772	15915	Small Database System

Table 2: Projects Used in the Experiments.

Project	Data	Correctness	Speed Test	Coverage
Quant	provided	✓	✓	85.71%
AStar	[Sturtevant, 2012]	✓	✓	83.17%
SkipList	provided	✓	✓	91.36%
AES	[Bell *et al.*, 1990]	✓	✓	93.69%
KGraph	Audio [Group, 2006], Sift1M [Jegou *et al.*, 2010]	✓	✓	-
MiniSQL	provided	✓	✓	83.73%

Table 3: Details of the correctness and performance testing.

optimizations produced by POLO surpass what the compiler can accomplish. We focus on the following questions:

RQ1: Overall Effectiveness. Is POLO capable of optimizing real projects, especially with the `O3` flag enabled?

RQ2: Baseline Compare. How does POLO compare to the existing state-of-the-art C++ program optimization methods?

RQ3: Ablation Study. How does the POLO perform without context or multiple rounds of agents' interactions?

RQ4: Case Study. What specific optimization techniques does POLO employ to optimize different programs?

4.1 Experiment Settings

Projects. The six C/C++ projects, summarized in Table 2, focus on algorithmic domains like SkipList and A* search, where performance is critical. They are classified into three levels based on lines of code: easy (≤ 500), medium ($> 500, \leq 2000$), and hard (> 2000), to assess the effectiveness in handling projects of varying scales. Notably, no existing dataset in project-level optimization. Related tasks—project-level code migration [Bairi *et al.*, 2024], code generation [Zhang *et al.*, 2023], and document generation [Luo *et al.*, 2024]—have used 6, 6, and 9 projects, respectively. Thus, we believe the number of projects in our study is reasonable, given the effort required for performance testing and correctness validation.

Correctness and Performance testing. Functional verification is a crucial step. The performance gains are meaningful only if it passes the functional tests of the origin version. Furthermore, it is necessary to provide domain-specific performance tests for each project. Table 3 presents the tests for each project. We also measured the code coverage of the correctness test program, which is above 80% in most cases.

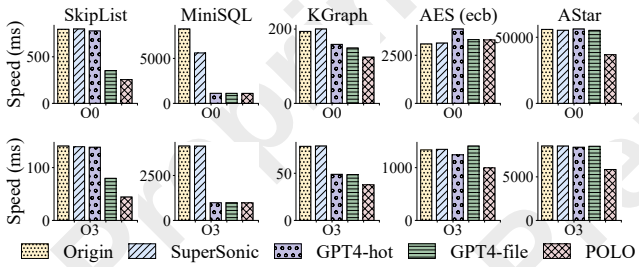

Figure 6: Comparison with baseline methods.

Evaluation metrics. Real-world projects have no absolute ground truth code. Therefore, we use the actual execution time of the program as the evaluation metric. Additionally, we also adopt the metric from the [Chen *et al.*, 2024], where the performance improvement (PI) is denoted as the ratio of the execution times of the origin ($origin$) and optimized version ($optimized$), defined as PI= $\frac{origin}{optimized}$. Each project is executed five times, and the average execute time is reported.

Implementation details. To balance effectiveness and cost, we use GPT-4o [OpenAI, 2024] as the default LLM agent. We set the (temperature: 0.2) to ensure the results are as deterministic and reproducible as possible. We use separate instances of the LLM as the Generator Agent and the Decision Agent to ensure role separation and avoid context entanglement. For each code optimization, we adopt a Top-N selection criteria, i.e., generating N results and selecting the best one among them. In our experiments, N=5. Occasionally, the LLM may generate compilation bugs, at which point we re-query the LLM to correct it. Our results show that the LLM does not produce a significant number of errors in optimization tasks. Because it refines the initial version of the code, rather than writing code from scratch. In the worst case, it generates code identical to the origin version, with no improvements.

4.2 RQ1: Overall Effectiveness

The object of RQ1 is to validate the effectiveness of POLO in real-world projects. For all projects, we conduct tests using both `O0` and `O3` flags. The goal is to assess whether the optimizations generated by POLO outperform those achieved by the compiler. The results in Table 4 demonstrate that POLO can significantly enhance project-level performance, achieving speedups ranging from several times to tens of times. Some optimizations even span across functions and files. We will further elaborate on the optimization through detailed case study. (Sec. 4.5). Notably, for data structures and algorithms projects (e.g., SkipList and AStar), the optimization effects are especially pronounced. For data processing and query-based projects (e.g., Minisql and KGraph), the optimized program demonstrates substantial performance gains across different data loads, including Audio and Sift1M datasets. In the Quant project, performance bottlenecks due to improper use of third-party libraries are addressed, leading to a significant improvement. For the AES project, POLO still achieves substantial improvements across various encryption modes.

4.3 RQ2: Baseline Compare

Unfortunately, we are unable to find an end-to-end system similar to POLO that spans from hotspot detection to optimization. Therefore, we manually inject the hotspot into baselines.

Supersonic [Chen *et al.*, 2024]: It uses a Seq2Seq model trained on pre- and post-optimization C/C++ code. Its training data is crawled from platforms like Codeforces, AIZU, and AtCoder. **GPT4-hot:** we use the more advanced GPT-4 Turbo [Achiam *et al.*, 2023], and send only the hotspot to the model as the prompt. **GPT4-file:** We provide the entire C++ file without explicitly marking the hotspot function to the GPT-4 Turbo. Notably, sending the hotspot for optimization to GPT-4 is a widely adopted baseline in this field, as seen in [Chen *et al.*, 2024; Garg *et al.*, 2023].

Projects		O0			O3		
		Origin (ms)	Optimized (ms)	PI (%)	Origin (ms)	Optimized (ms)	PI (%)
SkipList		794.6	255.5	**3.11**×	140.7	44.1	**3.19**×
Quant		671.9	30.9	**21.7**×	178.7	8.3	**21.5**×
AStar		55925.2	36853	**1.52**×	8527.6	5844.6	**1.46**×
MiniSQL		8260	1107.8	**7.45**×	4133.8	977.6	**4.23**×
KGraph	Audio	193.6	124.3	**1.56**×	78.5	37.8	**2.08**×
	Sift1M	38671.8	28422.2	**1.36**×	14222.5	9096.5	**1.56**×
AES	ECB	3083.3	3298.1	**0.93**×	1336.6	996.9	**1.34**×
	CBC	3136.7	3330.0	**0.94**×	1323.7	989.0	**1.34**×
	CTR	3141.1	3340.1	**0.94**×	1326.1	989.8	**1.34**×

Table 4: The overall effectiveness of POLO.

The results are shown in Fig. 6. Due to the simplicity of Quant's changes (Appendix B.1), the baseline's optimized code matches POLO's and is omitted from the figure. Our method consistently outperforms all baselines. Notably, Supersonic demonstrates minimal optimization across most projects, with only minor changes (e.g., adding `#pragma GCC optimize("unroll-loops")`) and no substantial function-level modifications. This limitation may stem from its training data, which is from Online Judge platforms, making it less suited for project-level optimizations. GPT4-hot can optimize function-level logic, but it becomes ineffective for project like SkipList, which require cross-function modifications. GPT4-file shows improvements in specific cases, especially when the hotspot and its context are within the same file, but it is limited for project like AStar, which require cross-file modifications.

4.4 RQ3: Ablation Study

Effect of multi-agent: Our framework relies on multi-round interactions between two agents to optimize the program. As shown in Table 5, we break down each individual optimization step and provide a detailed discussion of the changes. In the Quant project, significant optimization is achieved in the first round, and the optimized code is already highly efficient. Consequently, the Decision Agent opts to terminate the optimization process early. The MiniSQL project also achieved substantial optimization results in the first round, with only marginal improvements in the second round compared to the first. For all other projects, multiple rounds of optimization yield further improvements. In the second round of interactions, substantial gains are observed for SkipList and AES. During the third round, although execution times improve under `O0`, performance under `O3` declines. This is due to optimizations in these projects, such as inlining caller functions, which underperform compared to the compiler's native inlining strategies. The Decision Agent identifies this slowdown and rejects the optimization attempt. For AStar and KGraph, optimal results are achieved after the third round of interaction. Throughout multi-round optimization, an interesting pattern emerges: during the initial rounds, the LLM focuses on optimizing the internal logic of the hotspot. Afterwards, it gradually adopts a more global perspective, performing cross-function changes or utilizing more advanced instruction sets.

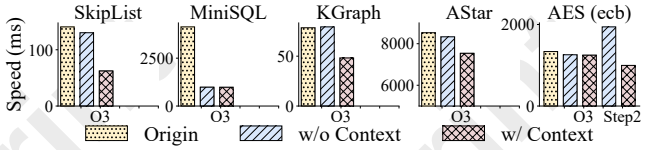

Figure 7: The impact of incorporating context information.

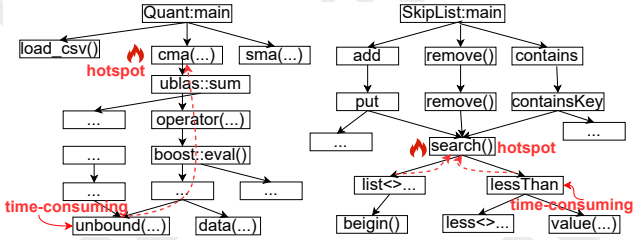

Figure 8: The runtime call graphs for the Quant and SkipList.

Effect of context: In Sec. 3.3, we synthesize neighboring nodes of the hotspot in the PSG to create a prompt. To evaluate the impact of context nodes, we compare two methods: one with ("w/ context") and one without the context ("w/o context") information. For a controlled comparison, we apply a single round of optimization, omitting the multi-round agent interactions. As shown in Fig. 7, "w/ context" performs similarly or better than the "w/o context" method. For intra-function optimizations like Quant and AES, the context-aware advantage is minimal, while for cross-function optimizations like SkipList and AStar, it significantly outperforms the "w/o context" method. In the AES project, both the "w/ context" and "w/o context" versions produce nearly identical optimized code after the initial optimization. To further refine both versions, we apply the multi-round agent strategy. The results are indicated by the Step 2 marker. Following this additional optimization, analysis reveals that the context-aware method achieves improvement by utilizing external function.

Our framework offers three key benefits: (1) Multi-agent and multi-round interactions enable more advanced optimization strategies, and (2) Decision Agent-environment interaction evaluates optimization effectiveness based on program feedback. (3) Context information is crucial for LLMs to

Projects		O0 (ms)				O3 (ms)			
		Step 0	Step 1	Step 2	Step 3	Step 0	Step 1	Step 2	Step 3
SkipList		794.63	508.73	255.54	**226.13**	140.67	62.23	**44.13**	50.69
Quant		671.90	**30.97**	-	-	178.70	**8.31**	-	-
AStar		55925.2	49189	43573.4	**36853**	8527.6	7536	7203.4	**5844.6**
MiniSQL		8260	1112.8	**1107.8**	1115	4133.8	989.8	**977.6**	985.4
KGraph	Audio	193.6	174.02	125.36	**124.25**	78.5	48.2	38.79	**37.80**
	Sift1M	38671.8	34622.4	28540.92	**28422.2**	14222.5	9835.45	9352.83	**9096.51**
AES	ECB	3083.29	3813.72	3298.06	**2759.26**	1336.59	1245.92	**996.87**	1110.29
	CBC	3136.65	3859.24	3330.01	**2758.77**	1323.74	1229.13	**989.04**	1092.31
	CTR	3141.04	3870.70	3340.12	**2782.09**	1326.12	1236.10	**989.81**	1091.39

Table 5: Results of each step in the multi-round interactions. Step 0 is the original program. The best results are highlighted in **bold**.

```
void SkipList<>::search(const K &value,
std::list<Node *> &prevList) {
    int h = head_->height-1;
    for(Node *p=head_;p!=nullptr;p=p->next[h]){
        if(lessThan(value,p)){
            p = p->prev[h];
            prevList.push_front(p);
            h--;
            if(h<0) break;
        }
    }
}

std::optional<V> SkipList<>::put(...) {
    std::list<Node*> list;
    search(key, list);
    ... // Details omitted
}
```

```
void SkipList<>::search(const K &value,
std::vector<Node *> &prevList) {
    int h = head ->height - 1;
    Node *p = head ;
    prevList.resize(head_->height, nullptr);
    while (h >= 0) {
        while (p->next[h] != nullptr &&
 !lessThan(value, p->next[h] )) {
            p = p->next[h];
        }
        prevList[h] = p;
        h--;
    }
}

std::optional<V> SkipList<>::put(...) {

    std::vector<Node*> list;
    ... // Remains the same
}
```

```
void SkipList<>::search(const K &value,
std::vector<Node *> &prevList) {
    ... // Remains the same
}

std::optional<V> SkipList<>::put(...) {
    static thread_local std::vector<Node*> list;
    ... // Remains the same
}

std::optional<V> SkipList<>::get(...) {
    static thread_local std::vector<Node*> list;
    ... // Remains the same
}

bool SkipList<>::remove(...) {
    static thread_local std::vector<Node*> list;
    ... // Remains the same
}
```

Figure 9: Case study of SkipList.

devise comprehensive project-level optimization strategies.

Effect of hotspot detection: We use Quant and SkipList as case studies to demonstrate the effectiveness of Func_Rank in identifying true hotspot functions. The key question we address is whether Func_Rank outperforms a naive baseline that simply selects the most time-consuming function as the hotspot. As shown in Fig. 8, the function with the longest execution time often lies on the call path to the actual hotspot, but is not the hotspot itself. In contrast, our approach analyzes the full call chain to accurately localize the true bottleneck.

4.5 RQ4: Case Study

This section illustrates how POLO optimizes projects step by step, complementing quantitative results with process-level insights. For additional case studies, please see the Appendix B.1.

SkipList: Fig. 9 shows the optimization in SkipList. In Step 1, we replace `std::list` with `std::vector`, as the latter's contiguous memory allocation improves cache locality, reducing cache misses. Modern processors' prefetching mechanisms further enhance cache hit rates. The internal loop logic is also optimized. In Step 2, due to the high frequency of calls to `put` and `get`, the cumulative cost of repeatedly allocating and deallocating memory for the temporary `std::vector<Node*>` becomes a performance bottleneck. By declaring the vector as `static thread_local`, memory is allocated only once and reused across function calls, effectively eliminating unnecessary allocation overhead.

AStar: Fig. 14 (see Appendix B.1) shows that the initial code frequently allocates and deallocates memory with each call to `getNeighbors`, as it creates a new `std::vector` to retrieve neighboring nodes for movement cost calculations. In Step 1, the function is modified to use a fixed-size array, returning only the count of valid neighbors, thus eliminating dynamic memory overhead and improving performance. Further optimizations include preallocating space in the `unordered_map` using `reserve` to reduce resizing costs, encapsulating priority queue operations in lambda functions for modularity, and inlining `getNeighbors` to decrease function call overhead. We use objdump disassembly tool to confirm that the previous code does not inline this function even under `O3` optimization, demonstrating the effectiveness of the manual inlining.

5 Conclusion

We present POLO, a novel framework for project-level program optimization. It identifies performance bottlenecks through dynamic analysis and a finite state machine that constructs the program call graph, followed by a function ranking algorithm to locate hotspots. Comprehensive static analysis extracts contextual information around these key elements. Subsequently, we propose a collaborative multi-agent method, where specialized agents interact over multiple rounds to iteratively improve performance based on feedback. Experiments on real-world projects demonstrate that POLO achieves significant speedups, even on top of the `-O3` optimization level.

Acknowledgments

This work has been supported by Zhejiang Province "Jianbing" Key R&D Project of China (No.2025C01010).

References

[Achiam *et al.*, 2023] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[Bairi *et al.*, 2024] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[Bell *et al.*, 1990] Timothy Bell, John Cleary, and Ian Witten. Calgary corpus. https://www.data-compression.info/Corpora/CalgaryCorpus/, 1990. Accessed: October 20, 2024.

[Chen *et al.*, 2024] Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in c/c++. *IEEE Transactions on Software Engineering*, 2024.

[Conte, 2015] Brad Conte. Basic implementations of standard cryptography algorithms, like aes and sha-1. https://github.com/B-Con/crypto-algorithms/tree/master, 2015. Accessed: October 20, 2024.

[DBAIWangGroup, 2021] DBAIWangGroup. Benchmark of nearest neighbor search on high dimensional data. https://github.com/DBAIWangGroup/nns_benchmark, 2021. Accessed: October 20, 2024.

[Della Toffola *et al.*, 2015] Luca Della Toffola, Michael Pradel, and Thomas R Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. *ACM SIGPLAN Notices*, 50(10):607–622, 2015.

[Du *et al.*, 2024] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[Gao *et al.*, 2024] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. Search-based llms for code optimization. *arXiv preprint arXiv:2408.12159*, 2024.

[Garg *et al.*, 2022] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. Deepperf: A deep learning-based approach for improving software performance. *arXiv preprint arXiv:2206.13619*, 2022.

[Garg *et al.*, 2023] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077*, 2023.

[Gregg, 2016] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[Group, 2006] Princeton CASS Group. Audio datasets. https://www.cs.princeton.edu/cass/demos.htm, 2006. Accessed: October 20, 2024.

[Huang *et al.*, 2023] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[Jegou *et al.*, 2010] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[Liu *et al.*, 2024a] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[Liu *et al.*, 2024b] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 2024.

[Luo *et al.*, 2024] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 436–464, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[Nethercote and Seward, 2007] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[Ni *et al.*, 2023] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

[Nistor *et al.*, 2013] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*, pages 237–246. IEEE, 2013.

[OpenAI, 2024] OpenAI. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/, 2024. Accessed: October 20, 2024.

[Page *et al.*, 1999] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford infolab, 1999.

[Roziere *et al.*, 2023] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[Shypula *et al.*, 2023] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

[Smidt, 2012] Joseph Smidt. quant++: A c++ quantitative trading framework. https://github.com/jsmidt/quantpp/tree/master, 2012. Accessed: October 20, 2024.

[Song and Lu, 2017] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380. IEEE, 2017.

[Sturtevant, 2012] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[Team, 2007] The Clang Team. Libtooling. https://clang.llvm.org/docs/LibTooling.html, 2007. Accessed: October 20, 2024.

[Wang *et al.*, 2024] Yi Wang, Qidong Zhao, Dongkuan Xu, and Xu Liu. Purpose enhanced reasoning through iterative prompting: uncover latent robustness of chatgpt on code comprehension. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 6513–6521, 2024.

[Weidendorfer, 2012] Josef Weidendorfer. Cache performance analysis with callgrind and kcachegrind. In *8 VI-HPS Tuning Workshop*, 2012.

[Xiao *et al.*, 2013] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 90–100, 2013.

[Xu *et al.*, 2010] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–186, 2010.

[Yang *et al.*, 2023] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Tingting Han, and Taolue Chen. Exploitgen: Template-augmented exploit code generation based on codebert. *Journal of Systems and Software*, 197:111577, 2023.

[Yu *et al.*, 2024] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[Zan *et al.*, 2022] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*, 2022.

[Zhang *et al.*, 2023] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore, December 2023. Association for Computational Linguistics.

[Zheng *et al.*, 2023a] Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. Self-infilling code generation. In *Forty-first International Conference on Machine Learning*, 2023.

[Zheng *et al.*, 2023b] Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. Outline, then details: Syntactically guided coarse-to-fine code generation. In *International Conference on Machine Learning*, pages 42403–42419. PMLR, 2023.

[Zhu *et al.*, 2024] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 437–445, 2024.