PEACE: Towards Efficient Project-Level Efficiency Optimization via Hybrid Code Editing

Xiaoxue Ren[†], Jun Wan[†], Yun Peng[‡], Zhongxin Liu[†*], Ming Liang[§], Dajun Chen[§], Wei Jiang[§], Yong Li[§]
[†]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Zhejiang University, Hangzhou, China
[‡]The Chinese University of Hong Kong, Hong Kong, China
[§]Ant Group, Hangzhou, China
{xxren, 22451014, liu_zx}@zju.edu.cn, ypeng@cse.cuhk.edu.hk, {liangming.liang, chendajun.cdj, jonny.jw, liyong.liy}@antgroup.com

***Abstract*—Large Language Models (LLMs) have demonstrated significant capability in code generation, but their potential in code efficiency optimization remains underexplored. Previous LLM-based code efficiency optimization approaches exclusively focus on function-level optimization and overlook interaction between functions, failing to generalize to real-world development scenarios. Code editing techniques show great potential for conducting project-level optimization, yet they face challenges associated with invalid edits and suboptimal internal functions. To address these gaps, we propose PEACE, a novel hybrid framework for Project-level code Efficiency optimization through Automatic Code Editing, which also ensures the overall correctness and integrity of the project. PEACE integrates three key phases: dependency-aware optimizing function sequence construction, valid associated edits identification, and efficiency optimization editing iteration. To rigorously evaluate the effectiveness of PEACE, we construct PEACEXEC, the first benchmark comprising 146 real-world optimization tasks from 47 high-impact GitHub Python projects, along with highly qualified test cases and executable environments. Extensive experiments demonstrate PEACE's superiority over the state-of-the-art baselines, achieving a 69.2% correctness rate (pass@1), +46.9% opt rate, and 0.840 speedup in execution efficiency. Notably, our PEACE outperforms all baselines by significant margins, particularly in complex optimization tasks with multiple functions. Moreover, extensive experiments are also conducted to validate the contributions of each component in PEACE, as well as the rationale and effectiveness of our hybrid framework design.**

I. INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable performance in code intelligence tasks, particularly in code generation [1], [2], [3]. Given natural language descriptions, they are capable of generating syntactically correct and functionally meaningful code, thus facilitating automated software development and largely improving productivity. This could be demonstrated by the 89% pass@1 of the state-of-the-art (SOTA) LLM on the EvalPlus benchmark [4].

Despite LLMs' great success in code generation, code quality assurance, especially for code efficiency optimization, which aims to improve the time efficiency of functionally correct code, is an urgent yet challenging task for LLMs. In real-world software development, code efficiency is a critical factor that impacts system scalability, efficiency, and maintainability [5], [6], [7], [8]. By analyzing 2,000 top-starred popular open-source Python projects from GitHub, we find that 41.25% (825 out of 2,000) of the projects have issues explicitly related to code efficiency optimization, explicitly by keywords related to efficiency (i.e., efficiency, speedup, etc.) in their issue reports. It further demonstrates the urgent demand for automated solutions that assist developers in optimizing the code efficiency of projects.

There are some research efforts [9], [10], [11], [12], [7], [6], [13] being devoted to LLM-based code efficiency optimization. For example, Shypula et al. [9] propose a broad range of adaptation LLM-based strategies for code efficiency optimization, including retrieval-based few-shot prompting, chain-of-thought, as well as fine-tuning. Garg et al. [7] solve performance bugs by retrieving relevant instructions from a knowledge base of past bug fixes and using them to generate a prompt, which is fed into an LLM in a zero-shot manner. SBLLM [10] formulates code efficiency optimization as a search problem by integrating LLMs with evolutionary methods for iterative refinement. Huang et al. [13] propose a self-optimization process that leverages open-source LLMs to generate a high-quality dataset, which is then used to fine-tune LLMs for code efficiency optimization.

However, **a key limitation is that prior work overlooks the complex function interactions when conducting code efficiency optimization, which hinders generalization to real-world scenarios**. In real-world development, code efficiency optimization often involves understanding project-wide context and modifying multiple functions. To address this, we reformulate project-level efficiency optimization as a code editing task that targets multiple functions, rather than generating or repairing a single one. Prior work [14], [15] shows that leveraging associated edits effectively guides such multi-function modifications.

To implement code editing techniques for project-level code efficiency optimization, we need to determine what code to edit, when to edit it, and how to edit it:

- *What to edit:* First, it is essential to identify the functions to be edited for efficiency optimization. Although the target function is pre-defined before optimization, its efficiency

* Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00153

may be largely affected by other functions due to call dependencies. Therefore, it is usually necessary to edit multiple functions at the same time to improve the efficiency of the target function.

- *When to edit:* After we collect the functions to edit, we need to decide their editing order based on the call relationships. As some functions may not have direct call relationships, it is important to edit them in an appropriate order to avoid the change of one function negatively affecting the change of another function.
- *How to edit:* With an ordered sequence of functions to edit, we need to know how to edit each function one by one. Previous studies [14], [15] reveal that the history edit records in the project provide valuable insights about how the project was previously optimized and could be used to guide future edits. However, as a vast number of edit records accumulate over time in large-scale software projects, ***invalid associated edits could adversely affect optimization guidance*** **(C1)**. In addition, project-level information may be misleading in code efficiency optimization, as the implementations of some internal functions in the projects may be suboptimal. This indicates that ***only considering internal functions in projects may result in suboptimal efficiency optimization*** **(C2)**.

To implement the above workflow and solve the challenges of project-level efficiency optimization, we propose PEACE, a novel hybrid framework for **P**roject-level **E**fficiency optimization through **A**utomatic **C**ode **E**diting. Specifically, PEACE consists of three phases: **(1) Dependency-aware optimizing function sequence construction**. It focuses on identifying and ordering the functions to be optimized, addressing both the *what to edit* and *when to edit* problems. By analyzing the relevance between the target function and its caller and callee functions, we construct an optimizing function sequence to ensure efficiency improvements are applied in a consistent and dependency-aware manner. **(2) Valid associated edits identification**, which is designed to address **C1** in *how to edit* problem. This phase iteratively retrieves and filters historical edits to identify valid associated edits that offer meaningful guidance for optimization. By combining dependency analysis with LLM-based semantic assessment, we ensure that only relevant and beneficial historical edits are used to guide the optimization. **(3) Efficiency optimization editing iteration**, which is designed to address **C2** in *how to edit* problem. This phase iteratively edits the functions in the constructed sequence using a fine-tuned efficiency optimizer. The optimizer leverages both internal and external high-performance implementations to generate more efficient solutions. Through repeated iterations, it enhances the project's overall efficiency while guaranteeing correctness and stability.

Since existing benchmarks rarely focus on project-level efficiency optimizations and cannot evaluate our PEACE, we construct a benchmark for the project-level code efficiency optimization task, named PEACEEXEC. PEACEEXEC contains 146 optimization tasks collected from 47 popular Python GitHub projects, covering 80 single-function and 66 multi-function optimization tasks. Each optimization task in our PEACEEXEC consists of a target function for optimization, the corresponding executable project, a task prompt, the historical edits, and the test cases for evaluation. Note, the task prompt is extracted from efficiency optimization related commit messages. If no explicit task prompt exists, we use general efficiency optimization instructions as the default input. The historical edits are from previous commits, and the test cases are collected from the efficiency-related commits.

We evaluate the performance of PEACE with PEACEEXEC regarding both editing correctness and execution efficiency improvement by measuring pass@1, opt rate and speedup, respectively. Unlike existing function-level code optimization techniques, where the opt rate is measured by comparing the execution performance before and after optimization, our project-level tasks sourced from GitHub lack pre-optimization test cases. Inspired by prompt-based optimization techniques [16], [9], we leverage the strong optimization capabilities of LLMs (e.g., GPT-3.5, GPT-4o) and measure the opt rate by comparing the results to those produced by GPT-4o with instruction prompting. Then, we also compare the efficiency of various optimization methods with the ground truth (e.g., the original human-optimized patches) of our collected tasks. Note, following Peng et al. [8], we adopt CPU instruction count as a stable and consistent metric for evaluating code efficiency, offering a more reliable alternative to execution time.

Extensive experiments illustrate that our PEACE achieves 69.2% of pass@1 and +46.9% of opt rate compared to the GPT-4o instruction-prompting method, which outperforms all selected baselines (e.g., CoEdPilot [15], SBLLM [10], and DeepDev-PERF [5]). Additionally, our PEACE achieves 0.840 speedup, which indicates that PEACE's optimization performance is closest to that of human-written patches, outperforming all other baselines. Moreover, we conduct ablation studies to illustrate the effectiveness of different components of our PEACE. We then experiment to investigate the performance of the fine-tuning performance optimizer in our PEACE, which can explain the rationality of the hybrid design of our framework. We also validate that our PEACE is well-designed and can excel in both single-function and multi-function optimization tasks, demonstrating robustness across varying task complexities.

In summary, our contributions are as follows:

- We propose PEACE, a novel hybrid framework for project-level efficiency optimization via automatic code editing, ensuring the overall correctness and integrity of the project.
- We construct the first benchmark for project-level code efficiency optimization tasks (i.e., PEACEEXEC), which contains 146 optimization tasks from 47 popular Python GitHub projects, covering 80 single-function and 66 multi-function efficiency optimization tasks.
- Extensive experiments compare PEACE with SOTA baselines to evaluate correctness and efficiency on project-level optimization tasks. Furthermore, we validate the rationale

and effectiveness of our hybrid framework design through additional experiments.
- We open-source the replication package [17], including PEACEXEC, the source code of PEACE, and results.

II. MOTIVATION

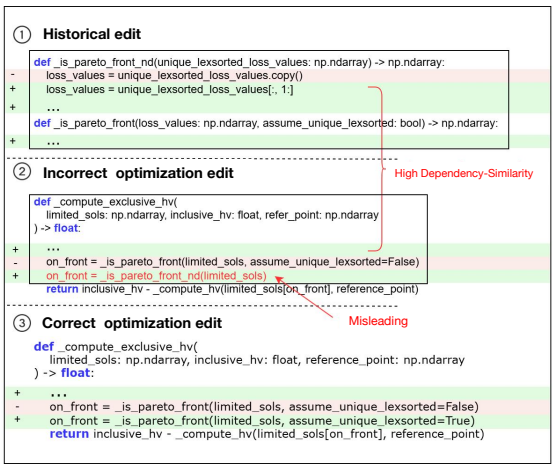

Fig. 1: Motivating Example of Valid Associated Edits

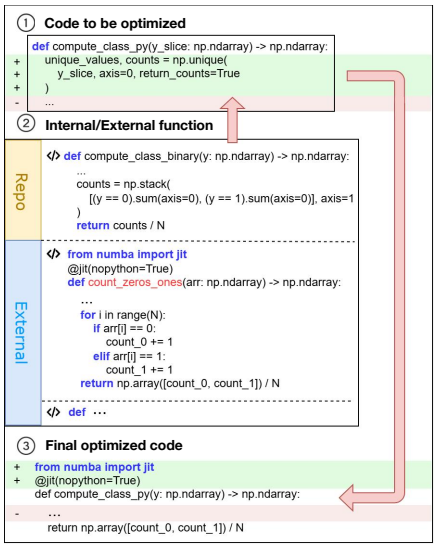

Fig. 2: Motivating Example of Optimization Knowledge

We use real-world examples to illustrate the two challenges mentioned in Sec I and motivate our work.

- **C1: Interference from invalid association editing.** In software projects, historical edits offer valuable guidance for future changes [14], [15]. However, using all historical edits is ineffective and impractical due to model input limits and noise. CoEdPilot [15] addresses this with a dependency score, but often struggles to filter out irrelevant/invalid edits, leading to incorrect or unnecessary modifications. As shown in Fig.1, the dependency-based method incorrectly identifies both `_is_pareto_front` and `_is_pareto_front_nd` as associated edits of the target function `_compute_exclusive_hv`, due to their high dependency similarity. If these misleading associations are incorporated into the editing process, the next edit may mistakenly select `_is_pareto_front_nd` (as shown in Fig.1 ②) rather than the correct `_is_pareto_front` (as shown in Fig.1 ③), leading to test failures and functional degradation.
- **C2: Limited optimization knowledge.** Existing project-level code intelligence tasks focus on internal code reuse and context within a project, which is effective for code generation but insufficient for optimization. Projects may repeatedly use suboptimal patterns that mislead optimization efforts. In contrast, high-performance functions from external projects can offer valuable insights for improving code quality. As shown in Fig. 2, the external function `count_zeros_ones` outperforms the internal function `compute_class_py` in terms of execution efficiency. Without incorporating external optimization knowledge, the final optimized function may result in suboptimal efficiency optimization.

Therefore, to address **C1**, it is essential to accurately identify valid associated edits that provide meaningful guidance for efficiency optimization, while effectively filtering out invalid or misleading candidates. To achieve this, we propose a method that combines dependency score analysis with semantic similarity assessment (see Phase II of PEACE). By jointly considering structural dependencies and semantic relevance, this approach significantly reduces the risk of introducing invalid edits and enhances the reliability of the optimization process. For **C2**, augmenting with internal and external optimization knowledge is crucial. This includes leveraging external high-performance implementations. To this end, we fine-tune an efficiency optimizer capable of generating diverse and efficient solutions. The optimizer integrates both internal project knowledge and external optimization resources, enabling more informed decision-making and achieving comprehensive, effective project-level code efficiency optimization (see Phase III of PEACE).

III. APPROACH

Fig. 3 shows the overall framework of PEACE, which aims to optimize the efficiency of a target function and ensure the correctness and integrity of the overall project at the same time. To achieve this goal, PEACE first analyzes code contexts to construct an optimizing function sequence for editing, and then identifies valid associated edits. After that, it leverages valid associate edits along with both internal and external high-performance functions to iteratively optimize the functions in the optimizing function sequence. Specifically, PEACE contains three main phases: Dependency-Aware Optimizing Function Sequence Construction (Phase I), Valid Associated Edits Identification (Phase II), and Code Efficiency Optimization Editing Iteration (Phase III).

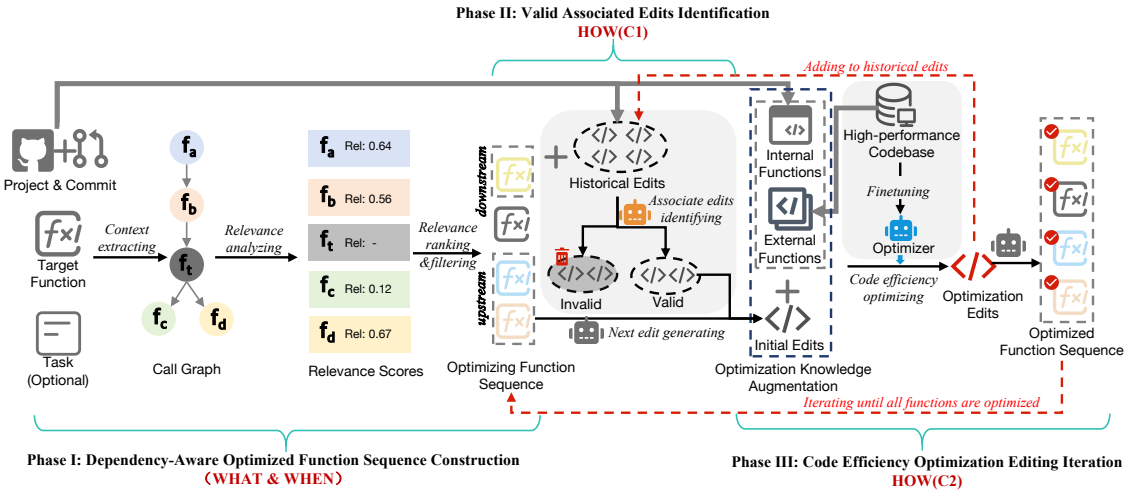

Fig. 3: The Overall Framework of PEACE

A. Dependency-Aware Optimizing Function Sequence Construction

A target function may depend on multiple related functions, requiring coordinated edits across them. In Phase I, PEACE builds a sequence of functions to optimize, starting from the target function, through three key steps.

1) Context Extraction: We first extract the optimization context by constructing a call graph of the target function to identify candidate function sequences. The call graph includes all callee functions directly/indirectly called by the target function and caller functions that directly/indirectly call the target function. Specifically, we parse code into an abstract syntax tree (AST) using tree-sitter [18]. Starting from the target function, we analyze function call nodes in the AST to identify its callees and callers, iterating over all functions to construct a complete call graph. For example, as Fig. 3 shows, the caller functions of target function f_t are f_a and f_b, and the callee functions include f_c and f_d.

2) Relevance Analysis: After obtaining the call graph of the target function, we further measure the relevance between the target function and its caller and callee functions. Following CoEdPilot [15], we employ a dual approach considering both structural and semantic similarities. Structural similarity captures dependencies of functions, while semantic similarity ensures functional relevance by filtering out syntactically related but semantically irrelevant functions. For dependency similarity, we utilize a transformer model (RobertaModel) in CoEdPilot [15] to estimate dependency scores between functions. For semantic similarity, we embed functions using the CodeBERT model and compute the cosine similarity. By combining dependency similarity with semantic similarity, we can obtain relevance scores between the target function and its caller and callee functions.

3) Relevance Filtering & Ranking: After computing the relevance scores between the target function and its caller and callee functions, we filter and rank these functions based on their scores to construct an optimizing function sequence, which serves as the edit order for project-level efficiency optimization. Specifically, we first remove the caller and callee functions with relevance scores below a defined threshold (i.e., 0.5, referred to CoEdPilot [15]), as they do not demonstrate sufficient structural and semantic relevance to the target function. For example, the callee function f_c is filtered out, as it gets a relevant score of 0.12. Then, we rank the remaining relevant functions in descending order based on their relevance scores. Note that following the bottom-up software development strategy [19], the optimized order is determined as follows: callee functions (e.g., f_d) are ranked before the target function (e.g., f_t), while caller functions (e.g, f_a and f_b) are ranked after the target function. For example, the final optimizing function sequence for editing in Fig. 3 is (f_d, f_t, f_a, f_b).

B. Valid Associated Edits Identification

As mentioned before, to deal with **C1**, we need to identify valid associated edits from the historical edits for each function to be optimized, avoiding misleading. The historical edits are collected from the previous commits of the corresponding GitHub projects. Before identifying, we first conduct a relevance analysis, following the approach in Sec III-A2, to measure the relevance between the function to be optimized and each historical edit. We can sort the historical edits in descending order of the relevance scores, which can efficiently filter out irrelevant edits that share no meaningful connection with the optimized function.

Then, we design an LLM-based agent to identify valid associated edits from the ranked historical edits. Fig. 4 illustrates the pipeline of our agent, which contains three steps:

- **Step 1: Task assignment.** In this step, we design a prompt with information from four aspects. The first aspect is the system prompt, which asks LLM to identify associated edits from our given ranked historical edits. Second, we prompt

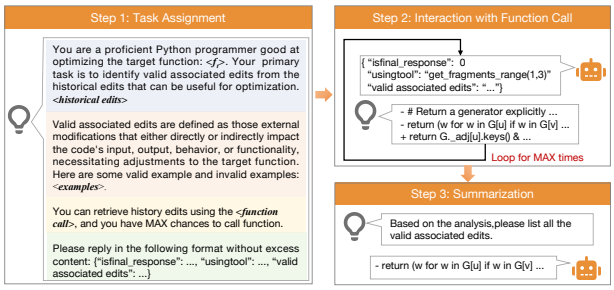

Fig. 4: Agent Pipeline for Identifying Valid Associate Edits

LLM with the definition of valid associated edits, which can help LLM develop a precise understanding. Third, we provide function calls for the LLM to use. At last, we restrict the output format.

- **Step 2: Interaction with function call.** We design a function call (i.e., `get_fragments_range(i,j)`) for LLM to interact with. So that LLM can prioritize retrieving historical edits with higher relevance and iterate sequentially. The iteration continues until the LLM determines that sufficient valid associated edits have been obtained or reaches a maximum number of iterations (i.e., 10).
- **Step 3: Summarization.** We instruct the LLM to summarize and list all identified valid associated edits by integrating inputs from Step 1 and the information retrieved in Step 2.

C. Code Efficiency Optimization Editing Iteration

1) Efficiency Optimizer Fine-tuning: To enhance the code efficiency optimization capability of general LLMs, we fine-tune an efficiency optimizer based on a small language model.

Before fine-tuning, we first construct a dataset specific to efficiency optimization tasks, which includes both function-level and project-level data. At the function level, we select optimization code pairs from the PIE dataset [9], which is constructed from performance-improving edits made by human programmers across various competitive programming tasks in CodeNet [20]. By filtering cases with performance improvements greater than 10% to ensure that only meaningful optimizations are included, we collected a total of 3,450 function-level cases. For the project level, we select 4,043 additional cases from GitHub, which are gathered through the benchmark construction process. These samples are performance-optimization related; however, due to limitations in execution feasibility, they are not included in the PEACExec. This fine-tuning dataset is available in our replication package.

After that, we employ LoRA [21] (Low-Rank Adaptation) to efficiently adapt the small language model to efficiency optimization tasks. During training, the model is presented with the original function alongside a set of semantically similar alternative implementations, as shown in Fig. 5 (left). Each implementation in the training set is accompanied by a performance benchmark to guide the model in identifying the best-performing version from a set of similar code. During inference, the model is instructed to generate the most efficient version of the function in terms of efficiency Fig. 5 (right).

This efficiency optimizer is then integrated into our PEACE to assist the LLM to generate the most efficient code given a set of candidate codes. Details are described in section III-C2

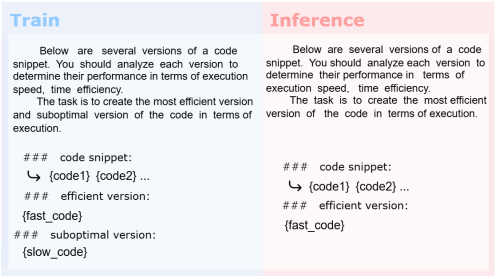

Fig. 5: Prompts for Fine-tuning Code Efficiency Optimizer

2) Code Efficiency Optimizing: As previously mentioned, both internal and external code efficiency optimization knowledge is essential for project-level efficiency optimization. To generate the most efficient edits for a project without compromising its correctness and integrity, we adopt the following process for optimization.

Specifically, for each function (e.g., f_d) in the optimizing function sequence (constructed in Phase I), (1) we first feed the function, along with its valid associated edits (identified in Phase II), into the LLM to generate an initial edit for optimization. (2) Next, we use the code functionality embedding model (i.e., TransformCode [22]) to retrieve functionally similar code snippets from both internal and external libraries. Here, internal libraries refer to code within the project itself, while external libraries consist of high-performance functions sourced from online repositories (i.e., Leetcode [23], Numba [24]), which are also available in our released package. (3) Then, the initial edit, along with the retrieved internal and external code snippets, is used to prompt our fine-tuned code efficiency optimizer, which generates an optimization edit. (4) At last, this optimization edit is integrated into the function by the LLM. This step is crucial, as the optimizer primarily focuses on efficiency improvements and may overlook correctness with respect to the broader project context. In this step, the LLM is prompted to incorporate the suggested edits while preserving contextual correctness. The detailed prompts can be found in our repository [17]. At the same time, the optimization edit is incorporated into the historical edits for the optimization of the next functions (e.g., f_t).

The iterative process continues until all functions in the optimizing function sequence have been optimized.

D. Implement

Our PEACE is a hybrid framework that integrates an LLM (i.e., Llama-3.1-405B) with a fine-tuned code efficiency optimizer based on Llama3-8B. We employ LoRA to fine-tune the optimizer, using a learning rate of 1×10^{-5} and a batch size of 16. The model is trained for 4 epochs to ensure sufficient exposure to optimization examples while avoiding overfitting. Additionally, the LoRA rank is set to 32.

IV. PEACEXEC CONSTRUCTION

As discussed earlier, previous studies on code efficiency optimization often primarily focus on function-level tasks, and existing datasets on project-level tasks [25], [26], [15], [5], [7] do not specifically address performance optimization or not available. To bridge this gap, we construct a project-level code optimization dataset, named PEACEXEC, from scratch.

A. PEACEXEC *Overview*

To assess the effectiveness of project-level code efficiency optimization techniques, our PEACEXEC is constructed based on GitHub issues related to efficiency optimization. Each project-level optimization task in our PEACEXEC involves the following components:

- **Target function:** It is treated as the entry function to be optimized. It does not mean that the task is reduced to a function-level optimization; rather, full contextual information is available, and it may be necessary to consider cross-function and cross-file edits to optimize the target function effectively.
- **Task prompt (optional):** It refers to the commit message of the efficiency optimization issues. It is optional because some commits do not explicitly describe the optimization intent. If no task prompt is available, our framework uses a generic optimization instruction instead.
- **Historical commit:** It is used to collect historical edits of the optimization task. As illustrated by previous work [14], [15], historical edits play important roles in predicting the next edits.
- **Executable project:** It is the corresponding project for optimization, along with its docker-based runtime environment.
- **Test case:** It is collected from the corresponding optimization issue commits and used for evaluations.

In total, our PEACEXEC contains 146 code efficiency optimization tasks covering 47 diverse projects, ensuring a comprehensive and representative assessment of project-level code efficiency optimization. Furthermore, we categorize the tasks in PEACEXEC into the following types:

- **Single-Function Project-Level Optimization:** It refers to simple optimization tasks that require editing only the target function itself to achieve project-level code efficiency optimization. This type consists of 80 tasks in our PEACEXEC.
- **Multi-Function Project-Level Optimization:** It refers to complex optimization tasks that require jointly editing multiple interdependent functions for project-level code efficiency optimization, while maintaining the consistency and functionality of projects. It covers 66 tasks in our PEACEXEC.

B. PEACEXEC *Construction*

Here is the pipeline of PEACEXEC construction, including project selection, optimization task filtering, project environment setup & test case validation.

1) Project Selection: Configuring executable environments for large-scale projects is challenging due to complex environment dependencies and unclear installation guidelines. To ensure that our benchmark closely aligns with real-world development scenarios, we carefully construct a dataset comprising over 2,000 of the top-starred Python projects from GitHub. Their varied coding styles, broad application domains, rich development patterns, and active maintenance contribute to a practical and scalable foundation for evaluating the effectiveness of techniques for project-level code efficiency optimization.

2) Candidate Optimization Task Filtering: Projects that have been maintained for an extended period often accumulate many issues and code commits over time. Among the selected projects, the most active project contains more than 200,000 commits. Considering the feasibility and effectiveness of projects, we first limited the candidate optimization tasks to the most recent 2,000 commits of each project, ultimately collecting approximately 440,000 candidate commits. After that, we continuously selected commits relevant to efficiency optimization, with the following steps:

- **Step 1: Coarse-grained filtering:** We first conduct a coarse-grained filtering by leveraging TF-IDF [27] to discover predefined efficiency-related keywords (e.g., "optimize", "latency", "efficiency" and "fast"), which are available in our released package [17]) in commits' messages. We filter out commits that do not contain such keywords, and reduce the number of candidates to around 16,000.
- **Step 2: Fine-grained filtering:** We then filter out commits that modify too many or too few lines/files from the 16,000 candidate commits. Specifically, we exclude commits that modify fewer than five lines, as these are considered trivial changes. Additionally, commits that modify more than 150 lines or involve five or more files are also removed, as they are deemed overly complex or redundant. Hence, 10,998 candidate commits remain.
- **Step 3: Semantic confirmation:** To simplify our task, we only focus on each commit having a single optimization goal, i.e., improving execution speed. The project-level code optimization is quite a difficult task in terms of the complexity of the projects themselves, so to make the optimization task more focused, we only cover the task of execution speed optimization in the dataset. This simplification of the optimization objective does not affect the practicality of our method, since we notice in the process of collecting dataset that the number of execution speed commits is much more than that of memory optimization commits. Specifically, for a more precise selection of performance-relevant commits, we employ the LLMs to make semantic confirmation of the remaining candidate commits. We utilize commit messages along with their corresponding code diffs to prompt LLMs to rank each commit based on its relevance to code efficiency. After this step, we retain 509 commits that show a strong correlation with code efficiency optimization issues, and such issues are candidate optimization tasks.

3) Project Environment Setup & Test Case Validation: After obtaining high-quality commits and corresponding candidate optimization tasks (i.e., issues) related to project-level code efficiency optimization, configuring executable environments and running test cases for evaluation presents a continued challenge. To ensure the reliability and feasibility of the selected tasks, we implement additional measures to validate that the test cases in commits are runnable and suitable for evaluating performance, along with ensuring the correctness of their respective environments.

- **Executable environment setup:** We create docker-based environments for each project to ensure proper setup of dependencies and configurations, enabling reliable code efficiency optimization.
- **Test case validation:** For each commit, we extract and prioritize test cases related to the function being optimized from commit patches. These test cases serve as essential evaluation metrics, ensuring that the code efficiency optimizations are verified under controlled conditions.

To further enhance the quality of our benchmark, we manually double-check to ensure the collected commits are related to efficiency optimization, and filter out cases lacking sufficient test case coverage. This rigorous selection process results in a total of 146 commits across 47 diverse projects. Then, we collect all components needed (i.e., target function, optional task prompt, historical commits, executable projects, and test cases), and construct our PEACEXEC for project-level code efficiency optimization. Note that the target function is manually selected based on the commit message and modification content.

V. EXPERIMENT SETUP

A. Research questions

To systematically validate the effectiveness of PEACE, we propose the following four research questions (RQs):

- **RQ1: What is the overall performance of our PEACE in dealing with project-level code efficiency optimization?** To answer RQ1, we systematically compare PEACE against popular baselines on PEACEXEC, aiming to investigate the capability of project-level code efficiency optimization.
- **RQ2: How does PEACE compare to baselines in terms of different optimization tasks?** For RQ2, we conduct a detailed comparative analysis of correctness and efficiency for PEACE and baselines across single-function and multi-function optimization tasks.
- **RQ3: What are the individual contributions of components designed in our PEACE, and how do they influence the overall performance?** For RQ3, we conduct an ablation study by replacing different components of PEACE with alternative implementations. This allows us to quantify the contribution of each component while also examining other factors that may influence performance in project-level code efficiency optimization tasks.
- **RQ4: To what extent can the fine-tuned code efficiency optimizer optimize the overall performance of PEACE? What is the significance and necessity of designing a hybrid framework?** For RQ4, we further investigate the capability of our fine-tuned performance optimizer by comparing it with popular proprietary LLMs. By replacing the optimizer with other models, we also investigate the design rationality of our hybrid framework.

B. Baselines

For overall performance evaluation, we select the following six relevant techniques as baselines, which cover both code editing techniques and efficiency optimization techniques. Due to the limited availability of research focusing on project-level optimization, we select both project-level and function-level optimization techniques as follows for evaluation.

- **Instruction-Prompting** [28]: We directly prompt the LLM to improve the performance of the given target function, GPT-4o is used as the inference model.
- **Fine-Tuning** [9]: It refers to a method that enhances pre-trained LLMs for code optimization by fine-tuning on performance-annotated datasets. We adopt the performance-conditioned generation strategy proposed by Shypula et al. [9].
- **SBLLM** [10]: It is a search-based framework that iteratively refines LLM-generated code optimizations. Specifically, in Stage 1 of SBLLM, we set $ns = 3$, retaining all three generated candidates for downstream processing and ultimately selecting the best-performing one. In Stage 2, we utilize the SBLLM-provided knowledge base for retrieval. In Stage 3, GPT-4o is employed as the inference model.
- **DeepDev-PERF** [5]: DeepDev-PERF is a deep learning-based method for software performance optimization proposed by Garg et al. [5]. Since the model and training data are not publicly released, we follow the training method described in their paper and use the dataset constructed by us in Section III.C of this paper to train a new model for comparison. Specifically, we extract statements, class attributes, caller-callee relationships, and signatures from the training data, and use these features to fine-tune a BART-large model for code generation.
- **RAPGen [7]**: RAPGen is a method that fixes performance bugs in software by first retrieving an instruction from a knowledge-base of previous fixes. Its core technology uses this retrieved prompt to guide an LLM in a zero-shot setting to generate an effective code change. Since RAPGen does not release both source code and the retriever model, we reproduce the RAPGen according to their paper with the dataset in our dataset in Section III.C.
- **CoEdPilot** [15]: It is a project-level code editing framework that integrates edit localization and feedback refinement, enabling context-aware code modifications based on prior edits. We adopt the open-source model provided by CoEdPilot [15], with the optimization task specified via prompts.

To conduct an ablation study for investigating contributions of different components of PEACE, we design the following variants by replacing different components of PEACE with alternative implementations.

- **PEACE _w/o_OFS:** It is designed to evaluate the performance of the optimized function sequence (OFS) constructed in phase I of PEACE. Specifically, we ablate the optimized function sequence from two aspects for evaluation: ① using the target function as the only optimized function without considering other relevant functions; ② considering multiple relevant functions but with a random sequence for efficiency optimization.
- **PEACE _w/o_VAE:** It is designed to investigate the contribution of valid associated edits (VAE), which are identified in phase II of PEACE. Specifically, we ablate the valid associated edits with two alternatives: ① no associated edits are given as an optimization assistant; ② associated edits extracted by the dependency analyzer [29] are given as an assistant of optimization.
- **PEACE _w/o_OKA:** It is designed to investigate the contribution of optimization knowledge augmentation (OKA) in phase III. Specifically, we consider three settings for this variant: ① no function augmentation and only use the initial generated edits for efficiency optimization editing; ② use external functions to augment initial edits for efficiency optimization editing; ③ use internal functions to augment initial edits for efficiency optimization editing.

Furthermore, in RQ4, we also replace our fine-tuned code efficiency optimizer in phase III with some popular LLMs to verify the performance of our fine-tuned optimizer and the necessity of this hybrid framework design. Specifically, the alternatives include open-source and closed-source LLMs with large parameters (i.e., Llama-3.1-405B [30], DeepSeek-V3-671B [31], GPT-4o [1], and Claude3.5 [32]), and one fine-tuned small LLM (i.e., CodeLlama-13B [33]).

C. Metrics

We employ the following metrics to assess the correctness and efficiency of the optimized functions, respectively:

- **Pass@1:** It represents the proportion of solutions that the model generates that correctly pass the test cases on the first try, which serves as an indicator of the correctness of the code editing [4].
- **Opt rate:** Prior research [8] demonstrates that CPU instruction count is a more stable metric than execution time for evaluating code efficiency. Consequently, we use the opt rate to quantify the relative change in CPU instruction consumption before and after optimization. Since project-level code optimization may introduce test case inconsistencies before and after modification, we define opt rate as the instruction count ratio relative to the baseline (i.e., instruction-prompting with GPT-4o), capturing the performance impact of different optimization strategies. Specifically, it is can be computed as $Opt\ Rate = \frac{I_{\text{instruction}} - I_{\text{method}}}{I_{\text{instruction}}}$, where $I_{instruction}$ represents the CPU instructions executed by the Instruction-Prompting baseline, I_{method} denotes the CPU instruction count of the evaluated optimization method.
- **Speedup:** The ratio $\frac{gt}{o}$ of CPU instruction count of ground truth solution (i.e., human-written patch) gt to the CPU instruction count of a code solution o. A speedup value exceeding 1.0 signifies superior performance of the generated code compared to the human benchmark. Conversely, a value approaching 1.0 from above indicates that the automated optimization's performance is converging on human-level quality. The primary objective is to maximize this value, striving for generated code that meets or exceeds human performance.

TABLE I: Overall Performance of PEACE in Project-level Code Efficiency Optimization

Method	PEACEXEC		
	Pass@1	Opt Rate	Speedup
Instruction-Prompting	52.7	-	0.446
Fine-Tuning	48.6	+24.5	0.591
DeepDev-PERF	34.9	+11.6	0.505
RAPGen	41.1	+12.2	0.508
SBLLM	58.2	+28.5	0.624
CoEdPilot	45.9	+7.8	0.484
PEACE	**69.2**	**+46.9**	**0.840**

VI. EVALUATION

A. Overall Performance of PEACE (RQ1)

For RQ1, we conduct experiments on PEACEXEC to show the overall performance of PEACE in project-level code efficiency optimization compared to the other baselines. We select six baselines, including three important code optimization methods (i.e., SBLLM [10], Instruction-Prompting with GPT-4o and Fine-Tuning with CodeLlama proposed by Shypula et al. [9]), three code performance optimization methods (i.e., Deepdev-PERF [5], RAPGen [7], SBLLM [10]), and the SOTA code editing method (i.e, CoEdPilot [15]). We evaluate the performance from both correctness and efficiency by measuring pass@1, opt rate & speedup, respectively. Note that (1) Test cases that do not pass the correctness checks are excluded from opt rate & speedup computation. (2) We consider Instruction-Prompting as the performance standard for measuring the opt rate of each method, as it is the most direct method leveraging the capabilities of GPT-4o, the most popular and SOTA LLM. Table. I shows the overall performance of our PEACE and baselines in project-level code optimization.

1) Correctness: As shown in Table I, our PEACE achieves a pass@1 correctness rate of 69.2%, significantly outperforming all baselines by 11.0% to 23.3%. This demonstrates that PEACE can generate accurate, reliable optimization edits without compromising correctness or project quality.

SBLLM delivers the second-best pass@1 score (58.2%) due to its broader search space, enabling exploration of diverse optimization candidates. However, its lack of fine-grained function dependency modeling limits its correctness. For example, when optimizing `_compute_exclusive_hv` (Fig. 1 ②), SBLLM introduces a pruning strategy that skips Pareto computation but fails to account for parent function dependencies, leading to incorrect results. DeepDev-PERF, RAPGen, and CoEdPilot perform the worst, with 37.8%, 41.1%, and 45.9% correctness. Although DeepDev-PERF and RAPGen is designed

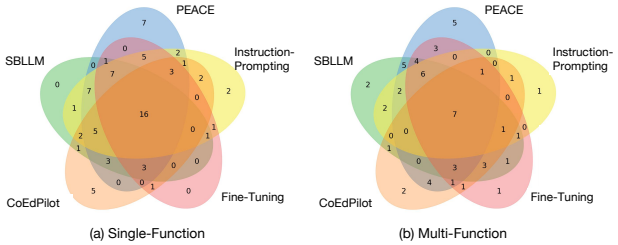

Fig. 6: Correctness Overlap Between PEACE and Baselines

for project-level code optimization, it struggles with complex function interactions. Additionally, CoEdPilot often introduces invalid edits when conducting project-level code editing. As shown in Fig. 1, it mistakenly replaces `_is_pareto_front` with `_is_pareto_front_nd`, causing computational errors due to incorrect dependency-based retrieval. Instruction-Prompting, using GPT-4o, achieves 52.7%. Its prompt-only approach limits its ability to model cross-functional dependencies, leading to incomplete or incorrect edits. Fine-Tuning performs slightly worse (48.6%), as its limited generalization hampers handling complex optimization tasks.

2) Efficiency: As shown in Table I, our PEACE achieves an impressive opt rate of +46.9%, surpassing all baselines by 18.4% to 39.1%. This highlights PEACE's ability to generate optimized code that significantly improves project-level runtime performance. Additionally, our PEACE achieves 0.840 in speedup, which indicates that PEACE's optimization performance is closest to that of human-written patches, outperforming all other baselines. The efficiency gains stem from PEACE's integration of both internal and external optimization knowledge. For example, when optimizing `compute_class_py` (Fig. 2), it retrieves `compute_class_binary` (internal) and `count_zeros_ones` (external, Numba JIT optimized), ultimately selecting the most efficient implementation.

In comparison, SBLLM achieves an opt rate of +28.5% by exploring diverse optimization candidates, ranking the second. However, its limited modeling of function dependencies and execution contexts constrains further improvements. DeepDev-PERF delivers +11.6% opt rate and Fine-Tuning delivers +24.5% opt rate, which shows that the lack of deeper analysis and optimization heuristics limits their capability in complex, repository-scale optimization scenarios. CoEdPilot shows minimal improvement (+7.8%), as it lacks mechanisms to target performance bottlenecks effectively.

Answer to RQ1: Our PEACE outperforms all baseline methods, achieving a high correctness of 69.2% pass@1 and delivering a substantial opt rate of +46.9% and speedup of 0.840. It highlights PEACE's effectiveness in improving both the correctness and efficiency of project-level code efficiency optimization.

B. Comparative Analysis (RQ2)

To further evaluate optimization performance, we conduct a comparative analysis from both correctness and efficiency aspects. Note that, for clarity of presentation, we select the five methods with the highest optimization correctness for further comparison, including our PEACE, SBLLM, Instruction-Prompting, Fine-Tuning, and CoEdPilot.

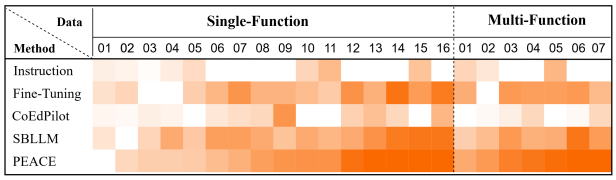

Fig. 7: Efficiency Comparison Between PEACE and Baselines on Correct Cases

1) Correctness: Figure 6 compares the correctness of PEACE and baseline methods in project-level optimization. The analysis covers two task types: single-function and multi-function, providing valuable insights into how PEACE compares to the baselines.

For single-function tasks shown in Figure 6 (a), PEACE demonstrates a significant advantage over the baselines, successfully optimizing seven unique tasks. While there is notable overlap between PEACE and SBLLM, it is clear that PEACE outperforms SBLLM, particularly in optimizing certain single-function tasks where SBLLM falls short. For multi-function tasks, as shown in Figure 6 (b), PEACE again shows a clear advantage over the baselines, excelling in optimizing complex tasks with high correctness across five unique tasks. While there is substantial overlap between PEACE and CoEdPilot, indicating that CoEdPilot is a strong baseline for performing correctness edits at the project level, it does not outperform PEACE in optimizing multi-function tasks.

2) Efficiency: For the efficiency comparison, we focus on the 23 optimization tasks that all methods successfully edit, including 16 single-function tasks and 7 multi-function tasks. Figure 7 uses color intensity to represent optimization performance, where darker shades indicate higher performance improvements. For single-function tasks, PEACE consistently outperforms all baseline methods. Figure 7 demonstrates that PEACE achieves the highest Opt rates in several tasks, particularly when compared to Instruction-Prompting and CoEdPilot, which show more sporadic improvements. While baselines like SBLLM and Fine-Tuning show some improvement in specific tasks, their performance is generally less consistent and lower than that of PEACE. For multi-function tasks, PEACE shows a significant advantage over baselines. Although CoEdPilot and SBLLM show some degree of improvement, their performance fails to match the high level achieved by PEACE. This underscores PEACE's ability to handle more complex optimization tasks effectively, where other methods struggle or exhibit less substantial improvements.

Answer to RQ2: PEACE outperforms baselines in both correctness and efficiency across single-function and multi-function optimization tasks, achieving better results, especially in complex tasks.

C. Ablation Study (RQ3)

To assess the impact of key components in PEACE, we perform an ablation study by systematically removing or modifying them. We construct three variants: PEACE _w/o_OFS, PEACE _w/o_VAE, and PEACE _w/o_OKA. Each variant includes two or three alternative versions of the removed component, marked as ①, ②, and ③ (see Sec V-B for details). We evaluate all variants on correctness (pass@1) and efficiency (opt rate & speedup). Results are shown in Table II.

1) Ablation of OFS: For optimizing function sequences (OFS), limiting edits to a single target function (PEACE _w/o_OFS_①) slightly improves correctness (70.5% vs. 69.2%) due to a smaller editing scope, but significantly reduces the opt rate to +29.5%, and the speedup drops to 0.633, limiting global performance gains. Allowing multi-function edits in random order (PEACE _w/o_OFS_②) raises the opt rate to +42.7% but drops correctness to 56.2%, showing instability from uncontrolled edit order. These results highlight the need to coordinate function selection and edit order for effective optimization.

2) Ablation of VAE: Removing historical edit information (PEACE _w/o_VAE_①) significantly reduces correctness (47.2% pass@1), highlighting the importance of prior knowledge for consistent and accurate edits. Interestingly, the efficiency remains moderately high (+34.3% of opt rate and 0.679 of speedup), indicating that some performance gains are possible, though often at the expense of reliability. Using a dependency-based retrieval method [29] (PEACE _w/o_VAE_②) improves correctness to 59.6% and opt rate to +41.9%, but still falls short of PEACE. This suggests that structural dependencies alone are insufficient; semantic context is also crucial for identifying valid associated edits that truly support effective optimization.

3) Ablation of OKA: For optimization knowledge augmentation, using only initial edits without any internal or external knowledge (PEACE _w/o_OKA_①) maintains correctness (69.6%) but significantly reduces the opt rate to +18.4% and the speedup is only 0.547, showing that our augmentation of the optimizer mainly boosts efficiency. Using only external high-performance functions (PEACE _w/o_OKA_②) improves opt rate (+41.7%) but slightly lowers correctness (68.9%), indicating efficiency gains at the cost of some inconsistency. In contrast, relying only on internal functions (PEACE _w/o_OKA_③) achieves the highest correctness (71.9%) but reduces the opt rate to +24.3%, suggesting internal knowledge preserves consistency but limits optimization potential.

Answer to RQ3: Our PEACE is well-designed, and all components (i.e., optimized function sequences, efficient association edits, and optimized knowledge augmentation) can be integrated together to contribute to the superior performance of PEACE in both correctness and efficiency.

D. Performance of Optimizer (RQ4)

To assess the effectiveness of our hybrid optimization framework, which combines an LLM (i.e., Llama3.1-405B) with a fine-tuned small model (i.e., Llama3-8B), we conducted comprehensive experiments by replacing our fine-tuned performance optimizer with other models, including open-source and closed-source LLMs and another fine-tuned small model.

TABLE II: The Result of Ablation Study

Method	Alternative	Metric		
		Pass@1	Opt Rate	Speedup
PEACE _w/o_OFS	①	70.5	+29.5	0.633
	②	56.2	+42.7	0.778
PEACE _w/o_VAE	①	47.2	+34.3	0.679
	②	59.6	+41.9	0.786
PEACE _w/o_OKA	①	69.6	+18.4	0.547
	②	68.9	+41.7	0.765
	③	71.9	+24.3	0.589
PEACE	-	**69.2**	**+46.9**	**0.840**

TABLE III: Effectiveness of Our Code Efficiency Optimizer

Type	Model	Metric		
		Pass@1	Opt Rate	Speedup
Closed-source	Llama-3.1-405B	66.7	+32.9	0.665
	DeepSeek-V3	70.8	+40.5	0.750
	GPT-4o	71.2	+37.2	0.710
	Claude-3.5	69.8	+33.6	0.672
Fine-Tuned	CodeLlama-13B	65.8	+46.4	0.832
	Llama-3-8B (Our)	**69.2**	**+46.9**	**0.840**

According to Table III, our code efficiency optimizer (i.e., fine-tuned with Llama-3-8B) achieves a pass@1 correctness of 69.2% and delivers a +46.9% opt rate and 0.840 in speedup, outperforming both larger fine-tuned models and closed-source LLMs in execution efficiency. Table III also shows that while LLMs with large parameters, such as GPT-4o and DeepSeek-V3, achieve higher correctness (71.2% and 70.8% on Pass@1, respectively), they offer lower efficiency gains compared to our fine-tuned small models. {Specifically, Llama-3.1-405B achieves only +32.9% in opt rate and 0.665 in speedup, significantly lagging behind the +46.9% achieved by our optimizer and far away from the efficiency of human-written patches. It demonstrates the advantage of our hybrid framework, where LLMs ensure correctness and semantic consistency at the project level, while fine-tuned small models, optimized for performance, deliver superior execution efficiency. The integration of both model types allows our PEACE to balance correctness and efficiency at the same time.

Answer to RQ4: Our fine-tuned performance optimizer outperforms other baselines. By combining a fine-tuned LLaMA-3-8B with Llama-3.1-405B, our hybrid framework PEACE can leverage the small model's strength in performance optimization and the LLM's capability in code generation. This design provides an efficient and reliable solution for project-level code optimization, ensuring both high correctness and execution efficiency.

VII. DISCUSSION

A. Computational Costs and Hybrid Architecture Design

PEACE adopts a hybrid architecture instead of fine-tuning a large language model (LLM, Llama3-1405B) to balance correctness, efficiency, and cost. It combines two lightweight models, a pre-trained CodeBERT for relevance analysis in Phase I and a fine-tuned optimizer for edit refinement in Phase III, with two stages of LLM interaction for validation and optimization. Direct fine-tuning of a large LLM would be prohibitively expensive and unsuitable for iterative optimization. PEACE therefore relies on the LLM mainly for project-wide reasoning and dependency-aware editing, while a smaller fine-tuned model (Llama3-8B) handles performance-oriented optimization at lower cost. Both lightweight models run within seconds on an Nvidia A800, so the primary overhead comes from LLM interactions. This cost is controlled through relevance filtering in Phase I, which reduces token usage and model calls. Most tasks converge within five interaction rounds, and a hard cap of ten iterations prevents unnecessary overhead. These measures ensure that PEACE remains computationally practical while achieving improvements in correctness and efficiency.

B. Threads to Validity

1) Internal Validity: Our work has three main limitations. First, our evaluation relies on correctness (pass@1) and execution efficiency (opt rate). While pass@1 assesses code correctness, our opt rate differs from typical optimization tasks, it compares performance against GPT-4o, since our GitHub-sourced project-level tasks lack pre-optimization test cases. Second, the baselines include SBLLM (function-level), DeepDev-PERF (repo-level), CoEdPilot (code editing), and LLM-based methods (Instruction-Prompting, Fine-Tuning). Third, we use static analysis to construct call graphs, which may miss dynamic calls (e.g., via inheritance or polymorphism). While effective for capturing explicit dependencies, future work will explore integrating dynamic analysis to improve accuracy.

2) External Validity: Our benchmark, PEACEXEC, is built from popular, highly starred Python projects on GitHub. This ensures it reflects widely used open-source projects but may limit generalization to other programming languages, such as C++ or Java, or to less-maintained codebases. Additionally, although our experiments include both open-source and closed-source, the rapid evolution of LLM architectures and fine-tuning techniques could impact future results. Therefore, our findings should be understood within the context of the current generation of models.

VIII. RELATED WORK

A. Code Optimization

With the rise of large language models (LLMs), code optimization has advanced at both the function and project levels. At the function level, PIE [9] benchmarks LLMs for performance-aware code generation using prompting. Later works [34], [35] improve results through prompting and fine-tuning. Techniques like STOP [6] and retrieval-based models [36], [37], [38] enhance quality via feedback and external knowledge. SBLLM [10] adds search-based refinement, while others [39], [40] incorporate profiling and static analysis to guide optimization. At the project level, methods like DeepDev-PERF [5] fine-tune models with project-wide context. RAPGen [7] uses performance-related commits to retrieve optimization examples. CodeDPO [41] applies iterative self-validation, and COCOGEN [42] combines compiler feedback with context retrieval. Despite this progress, general-purpose, scalable solutions for project-level performance optimization remain limited, highlighting the need for further research.

B. Code Editing

Code editing has gained growing interest, with various methods developed to predict and generate code changes. Early approaches like Codit [43] used tree-based neural networks to model edits based on AST hierarchies. Recoder [44] improved this by combining AST and code readers to better understand code structure and boost prediction accuracy. Later methods explored pre-training and transformer models. CURE [45] used pre-trained models for program repair, showing the benefits of large-scale pre-training for producing correct edits. CoditT5 [46] extended CodeT5 [47] by training on inputs that include comments and code changes, generating structured edit plans for more accurate edits. Recent work focuses on prompt-based learning with large language models. GRACE [14] fine-tunes LLMs with prompts that include related code updates, improving edit quality. CoEdPilot [15] also uses LLMs to identify relevant prior edits and predict where changes should be made, enabling more precise and context-aware editing. Existing code editing methods focus on general edits, limiting their use in performance optimization. In contrast, we propose a hybrid framework tailored for targeted, effective code optimization.

IX. CONCLUSION

We propose PEACE, a hybrid framework for project-level performance optimization via automatic code editing. Unlike function-level approaches, PEACE captures interdependencies among functions to optimize multiple components while preserving correctness. Evaluated on PEACEXEC, including 146 tasks from 47 Python GitHub projects, our PEACE significantly outperforms state-of-the-art methods like CoEdPilot, SBLLM, and DeepDev-PERF in both correctness and efficiency. PEACE offers a robust solution for real-world software optimization, with future work aiming to extend language support and improve integration with external libraries.

ACKNOWLEDGMENT

This research/project was partially supported by the National Natural Science Foundation of China (No.62302437, No. 62202420), the Fundamental Research Funds for the Central Universities 226-2025-00004, and Zhejiang Provincial Natural Science Foundation of China (No.LZ25F020003).

REFERENCES

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[3] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[4] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[5] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "Deepdev-perf: a deep learning-based approach for improving software performance," ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 948–958. [Online]. Available: https://doi.org/10.1145/3540250.3549096

[6] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, "Self-taught optimizer (stop): Recursively self-improving code generation," in *First Conference on Language Modeling*, 2024.

[7] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "Rapgen: An approach for fixing code inefficiencies in zero-shot," *arXiv preprint arXiv:2306.17077*, 2023.

[8] Y. Peng, J. Wan, Y. Li, and X. Ren, "Coffe: A code efficiency benchmark for code generation," *ArXiv*, vol. abs/2502.02827, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:276116683

[9] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," *arXiv preprint arXiv:2302.07867*, 2023.

[10] S. Gao, C. Gao, W. Gu, and M. Lyu, "Search-based llms for code optimization," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 254–266.

[11] J. Gong, V. Voskanyan, P. Brookes, F. Wu, W. Jie, J. Xu, R. Giavrimis, M. Basios, L. Kanthan, and Z. Wang, "Language models for code optimization: Survey, challenges and future directions," *arXiv preprint arXiv:2501.01277*, 2025.

[12] D. Huang, G. Zeng, J. Dai, M. Luo, H. Weng, Y. Qing, H. Cui, Z. Guo, and J. M. Zhang, "Effi-code: Unleashing code efficiency in language models," *arXiv preprint arXiv:2410.10209*, 2024.

[13] D. Huang, J. Dai, H. Weng, P. Wu, Y. Qing, J. M. Zhang, H. Cui, and Z. Guo, "Soap: enhancing efficiency of generated code via self-optimization," *arXiv e-prints*, pp. arXiv–2405, 2024.

[14] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Language models meet code edits," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1483–1495.

[15] C. Liu, Y. Cai, Y. Lin, Y. Huang, Y. Pei, B. Jiang, P. Yang, J. S. Dong, and H. Mei, "Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 466–478.

[16] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, "Self-taught optimizer (STOP): Recursively self-improving code generation," 2024. [Online]. Available: https://openreview.net/forum?id=1gkePTsAWf

[17] linrenmeng. (2025) Peace: Towards efficient project-level performance optimization via hybrid code editing. Accessed on March 5, 2025. [Online]. Available: https://github.com/linrenmeng/Peace/tree/master

[18] treesitter. (2025) Tree-sitter. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[19] A. Hollberg, T. Lützkendorf, and G. Habert, "Top-down or bottom-up?–how environmental benchmarks can support the design process," *Building and Environment*, vol. 153, pp. 148–157, 2019.

[20] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. A. Zolotov, J. Dolby, J. Chen, M. R. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, and U. Finkler, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," in *NeurIPS Datasets and Benchmarks*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235195915

[21] J. E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *ArXiv*, vol. abs/2106.09685, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235458009

[22] Z. Xian, R. Huang, D. Towey, C. Fang, and Z. Chen, "Transformcode: A contrastive learning framework for code embedding via subtree transformation," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1600–1619, 2024.

[23] LeetCode. (2025) Leetcode. Accessed on March 5, 2025. [Online]. Available: https://leetcode.cn/

[24] N. Developers. (2025) Numba: High performance python compiler. Accessed on March 5, 2025. [Online]. Available: https://numba.pydata.org/

[25] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[26] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[27] S. Qaiser and R. Ali, "Text mining: use of tf-idf to examine the relevance of words to documents," *International journal of computer applications*, vol. 181, no. 1, pp. 25–29, 2018.

[28] S. Mishra, D. Khashabi, C. Baral, Y. Choi, and H. Hajishirzi, "Reframing instructional prompts to GPTk's language," in *Findings of the Association for Computational Linguistics: ACL 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 589–612. [Online]. Available: https://aclanthology.org/2022.findings-acl.50/

[29] W. Jin, D. Zhong, Y. Cai, R. Kazman, and T. Liu, "Evaluating the impact of possible dependencies on architecture-level maintainability," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1064–1085, 2023.

[30] H. Face. (2025) Llama-3.1-405b. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.1-405B

[31] Deepseek. (2025) Deepseek-v3. [Online]. Available: https://github.com/deepseek-ai/DeepSeek-V3

[32] Anthropic. (2025) Claude-3.5. [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[33] H. Face. (2025) Codellama. [Online]. Available: https://huggingface.co/codellama/CodeLlama-13b-hf

[34] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 31 693–31 715.

[35] S. Gao, W. Mao, C. Gao, L. Li, X. Hu, X. Xia, and M. R. Lyu, "Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 969–981, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266725647

[36] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, and Dhariwal, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[37] Y. Levine, I. Dalmedigos, O. Ram, Y. Zeldes, D. Jannai, D. Muhlgay, Y. Osin, O. Lieber, B. Lenz, S. Shalev-Shwartz, A. Shashua, K. Leyton-Brown, and Y. Shoham, "Standing on the shoulders of giant frozen language models," 04 2022.

[38] W. Shi, S. Min, M. Yasunaga, M. Seo, R. James, M. Lewis, L. Zettlemoyer, and W.-t. Yih, "REPLUG: Retrieval-Augmented Black-Box Language Models," *arXiv e-prints*, p. arXiv:2301.12652, Jan. 2023.

[39] C. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *ArXiv*, vol. abs/2304.00385, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:274610860

[40] S. Lu, N. Duan, H. Han, D. Guo, S. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," in *ACL 2022 - 60th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers)*, ser. Proceedings of the Annual Meeting of the Association for Computational Linguistics, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association

for Computational Linguistics (ACL), 2022, pp. 6227–6240, publisher Copyright: © 2022 Association for Computational Linguistics.; 60th Annual Meeting of the Association for Computational Linguistics, ACL 2022 ; Conference date: 22-05-2022 Through 27-05-2022.

[41] K. Zhang, G. Li, Y. Dong, J. Xu, J. Zhang, J. Su, Y. Liu, and Z. Jin, "Codedpo: Aligning code models with self generated and verified source code," 2024. [Online]. Available: https://arxiv.org/abs/2410.05605

[42] Z. Bi, Y. Wan, Z. Wang, H. Zhang, B. Guan, F. Lu, Z. Zhang, Y. Sui, H. Jin, and X. Shi, "Iterative refinement of project-level code context for precise code generation with compiler feedback," 2024. [Online]. Available: https://arxiv.org/abs/2403.16792

[43] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[44] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[45] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[46] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556955

[47] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *ArXiv*, vol. abs/2109.00859, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:237386541