

Investigating Execution-Aware Language Models for Code Optimization

Federico Di Menna[†], Luca Traini[†], Gabriele Bavota[‡], Vittorio Cortellessa[†]

[†]University of L'Aquila, L'Aquila, Italy

[‡]Software Institute - Università della Svizzera Italiana, Switzerland

federico.dimenna@graduate.univaq.it, luca.traini@univaq.it, gabriele.bavota@usi.ch, vittorio.cortellessa@univaq.it

Abstract—Code optimization is the process of enhancing code efficiency, while preserving its intended functionality. This process often requires a deep understanding of the code execution behavior at run-time to identify and address inefficiencies effectively. Recent studies have shown that language models can play a significant role in automating code optimization. However, these models may have insufficient knowledge of how code execute at run-time. To address this limitation, researchers have developed strategies that integrate code execution information into language models. These strategies have shown promise, enhancing the effectiveness of language models in various software engineering tasks. However, despite the close relationship between code execution behavior and efficiency, the specific impact of these strategies on code optimization remains largely unexplored. This study investigates how incorporating code execution information into language models affects their ability to optimize code. Specifically, we apply three different training strategies to incorporate four code execution aspects — line executions, line coverage, branch coverage, and variable states — into CodeT5+, a well-known language model for code. Our results indicate that execution-aware models provide limited benefits compared to the standard CodeT5+ model in optimizing code.

Index Terms—Code Optimization, Deep Learning

I. INTRODUCTION

Performance is a critical quality attribute of modern software systems [1]. Improving software performance involves activities across various layers of the software stack, ranging from architectural design [2] to compiler optimization [3]. Among these, source code optimization — the process of refining code by selecting efficient data structures and algorithms — stands out as a key approach for enhancing the efficiency of software systems.

As of today, code optimization tasks largely rest on the shoulders of developers. However, with the rising trend of using AI to automate software engineering tasks [4]–[9], recent studies have begun exploring the potential of language models to automate code optimization [10]–[13]. In these studies, language models are provided with a non-optimized version of a code snippet (e.g., a function) and tasked with generating an optimized version that improves specific performance properties, such as execution time.

While these efforts have demonstrated the significant potential of language models in automating code optimization [10]–

[13], these models are typically trained on a “static” source code representation and may therefore lack an understanding of how code executes at run-time [14]–[16]. Recent studies have shown that integrating these models with code execution information can significantly enhance their effectiveness across a variety of downstream software engineering tasks [15]–[18]. For instance, Ding *et al.* [18] proposed a pre-training strategy to teach language models specific aspects of code execution, such as branch coverage and variable states, demonstrating improvements in tasks like clone retrieval and vulnerability detection. Similarly, Ni *et al.* [15] introduced *NExT*, a method that enables language models to inspect variable states of executed code lines and reason about their execution behavior, resulting in a higher fix rate for program repair tasks. Despite these and other efforts [16], [17], [19]–[21], the impact of execution-awareness in automated code optimization remains largely unexplored.

Given the close relationship between run-time execution behavior and code efficiency, this paper investigates how teaching language models to understand code execution behavior affects their effectiveness in optimizing code. Specifically, we first train a CodeT5+ model [22] with training objectives related to four code execution aspects, namely number of line executions, line coverage, branch coverage, and variable states. We then evaluate and compare the code speed-ups achieved by these execution-aware models against those of the standard CodeT5+ model. Our findings reveal that execution-aware models do not outperform the traditional language model in code optimization. On the contrary, they seem to be less effective than the standard CodeT5+ model.

The contribution of this paper are as follows:

- A comprehensive evaluation of twelve execution-aware language models, based on CodeT5+, for code optimization. This evaluation encompasses four different code execution aspects and three distinct strategies.
- A complementary analysis providing insights to guide future research in automated code optimization.
- A replication package to reproduce our findings¹.

Paper Structure: Section II provides background on automated code optimization and execution-aware language models for code. Section III describes our study design and research questions. Section IV explains the experimental setup.

This article has been accepted for publication in *The 33rd IEEE/ACM International Conference on Program Comprehension (ICPC 2025)*, *RENE Track*. This version of the manuscript is a preprint and may differ from the final published version in terms of content and formatting.

¹<https://github.com/SpencerLabAQ/exec-aware-code-opt>

Section V reports the results. Section VI presents a qualitative analysis, discusses the results, and describes threats to validity. Section VII concludes this paper.

II. BACKGROUND AND RELATED WORK

A. Language Models for Code Optimization

Transformer-based language models are a category of deep learning models that have driven significant advancements in natural language processing. In recent years, studies have demonstrated the ability of such models to effectively perform various software engineering tasks [4]–[8], [23]–[27]. Since code optimization involves generating a refinement of a source code that improves efficiency, it can be directly framed as a code-to-code transformation task (specifically, from inefficient to efficient version). For example, Garg *et al.* proposed *DeepDev-PERF* [13], a transformer-based model pretrained on English and C# source code corpora, and further specialized with performance-improving commits. Their work shows that *DeepDev-PERF* can generate code changes that lead to tangible performance optimizations to various open source C# projects.

Shypula *et al.* proposed a dataset of performance-improving code edits (*PIE*) [10], composed of slow-fast C++ program pairs. They exploit this dataset to evaluate a variety of techniques, including fine-tuning and prompting, for adapting well-known language models, such as CodeLLama, GPT-3.5 and GPT-4, for code optimization. Their results show that certain combinations of techniques and models can achieve higher optimizations than human reference. Similarly, Chen *et al.* [28] proposes *SUPERSONIC*, a smaller seq2seq model targeting minor source code modifications for optimization. *SUPERSONIC* is trained on C/C++ slow-fast program pairs leveraging a diff-based input representation. Results show that it outperforms larger language models, such as GPT-3.5-Turbo and GPT-4, on code optimization while retaining significant similarity with the original program. RAPGen [12] leverages a pre-constructed knowledge-base to prompt language models in zero-shot to generate code inefficiencies fix. Gao *et al.* [11] modeled the optimization task with a search-based approach, integrating language models with evolutionary search and outperforming several baselines models such as CodeLLama, Gemini and ChatGPT.

Although these studies have demonstrated the effectiveness of language models in optimizing code, they tend overlook the potential implications of integrating code execution information into the model.

B. Execution-aware Language Models

Most existing language models for code focus on learning static form of code text, however they may lack a semantic understanding of how code execute at run-time [14], [15]. To mitigate this issue, several approaches have been proposed to integrate language models with code execution information. For instance, Ding *et al.* proposed *TRACED* [18], an execution-aware pre-training strategy to capture run-time execution aspects of code. This strategy pre-trains the language model

with the task objective of predicting variable states, and line coverage, forcing the model to reason about code execution behavior. They found that learning code execution behavior, significantly improves the capabilities of language models in both clone retrieval and vulnerability detection tasks.

Another example is *SemCoder* [16], an execution-aware language model that leverages monologue reasoning to learn about local execution effects of individual statements, overall input/output behavior, thereby linking static code with code execution behavior. *SemCoder* shows competitive effectiveness with GPT-3.5-turbo on code generation and execution reasoning tasks, despite being significantly (6.7B parameters). A similar approach, namely *NExT*, was recently proposed by Ni *et al.* [15]. This method teaches the language models to inspect the execution traces of programs (variable states of executed lines) and reason about their run-time behavior through chain-of-thought rationales. *NExT* considerably improves the fix rate of a PaLM 2 model for program repair tasks.

Huang *et al.* [17] proposed *FuzzPretrain*, a method to explore run-time execution information of code revealed by their test cases and embed it into the feature representations of code as complements. *FuzzPretrain* yields significant improvements on the code search task over its language model counterparts trained with only static source code.

Previous research has studied the effectiveness of execution-aware models across various software engineering tasks, such as program repair [15], vulnerability detection [18], clone retrieval [18], code generation [16], and code search [17]. However, despite the clear connection between runtime execution behavior and code efficiency, no prior work has directly examined how incorporating code execution information into language models influences their ability to optimize code. This paper seeks to address this gap.

III. STUDY DESIGN

The *goal* of this study is to investigate how integrating code execution information into language models impacts their ability to optimize code. We analyze twelve execution-aware models, derived from CodeT5+, that incorporate different combinations of four code execution aspects and three training strategies. Specifically, for each code execution aspect, we create execution-aware models using three distinct training strategies. We then evaluate the code optimizations provided by these execution-aware models and compare them to those produced by the standard CodeT5+ model.

In the following sub-sections, we describe the code execution aspects and training strategies investigated in this study, as well as the research questions we aim to address.

A. Code Execution Aspects

1) *Line Executions (LE)*: This aspect refers to how often a specific line of source code is executed during runtime. Each line of code is assigned a label indicating the number of times it has been executed. This information provides understanding on *hotspots* within the code, *i.e.*, code sections that are frequently executed and may benefit from optimization.

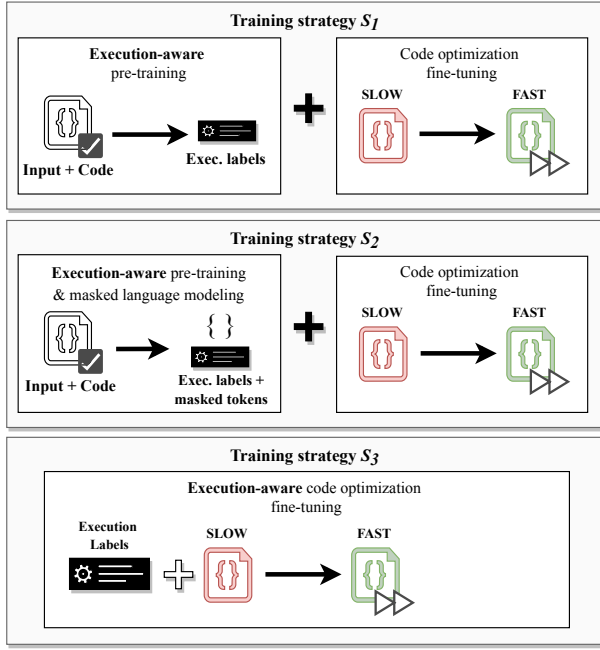


Fig. 1: Overview of the proposed training strategies for building execution-aware language models.

2) *Line Coverage (LC)*: Line coverage identifies which lines of source code are executed at least once during execution. This provides insights about which parts of source code are covered after executing the program with a specific input, and which one remain underutilized.

3) *Branch Coverage (BC)*: This aspect is primarily designed to determine the parts of the source code associated to a branch. Given the execution of the program with a specific input, each line of code can fall into one of three categories: (i) it is part of a covered branch, (ii) it is part of an uncovered branch, or (iii) it is not part of any branch.

4) *Variable States (VS)*: This information focuses on how the code execution changes the state of variables. For example, some variables may control the program's flow (e.g., counters or boolean flags), while others store intermediate or final results to be returned. Understanding the state of these variables at the end of execution provides insights into the code behavior at runtime.

B. Training Strategies

We define three distinct training strategies, each applied to a language model which has been already pre-trained on a static representation of code text, specifically CodeT5+. Each strategy is designed to teach the model a particular aspect of code execution using a specific training methodology. Figure 1 presents a high-level overview of the stages involved in each training strategy. In particular, we consider the following strategies: (i) *execution-aware pre-training* (S_1), (ii) *execution-aware pre-training combined with masked language modeling* (S_2), and (iii) *execution-aware fine-tuning* (S_3). In the following, we provide a description of each training strategy, while

their instantiations to specific training datasets are detailed in Section IV-A.

S_1 — *Execution-aware Pre-training + Fine-tuning*: This strategy involves two sequential training stages. In the first stage, the model undergoes pre-training to predict aspects of code execution based on specific input test cases. To deepen the model understanding of runtime code execution, we train it on multiple input cases to capture diverse execution behaviors of the same code. We term this process *execution-aware pre-training*. Specifically, the model learns to predict run-time execution information without actually running the code (i.e., it is provided with a program and a specific input and it must predict a specific execution aspect, like for example the line coverage). This step is designed to teach the language model how code executes at runtime. In the second stage, the execution-aware model is fine-tuned for code optimization. This is framed as a sequence-to-sequence downstream task, where the input is a slower version of a program, and the output is an optimized, faster version. Importantly, the optimization process aims to preserve the program's intended functionality; that is, the optimized program must produce the expected output.

S_2 — *Execution-aware Pre-training & Masked Language Modeling + Fine-tuning*: Ding et al. [18] demonstrated that combining execution-aware pre-training with *masked language modeling* (MLM) can enhance model effectiveness across multiple software engineering tasks. Building on this approach, this strategy combines an execution-aware objective (as the one described for S_1) with an MLM objective during pre-training. The MLM objective involves randomly masking 15% of tokens in the input sequence and training the model to predict the masked tokens based on the surrounding context. Following the pre-training stage, as in the S_1 strategy, the model is fine-tuned for code optimization.

S_3 — *Execution-aware Fine-tuning*: This strategy does not involve a pre-training stage. Instead, code execution information is incorporated directly into the model input (i.e., the slow version of the code) during fine-tuning. This information is added either line-by-line (for *LE*, *LC*, and *BC*) or at the end of the code (for *VS*). Unlike the first two strategies, this one assumes the availability of an execution trace for the input code. Since code execution information is typically tied to a specific input test case, we adopt different methods depending on the aspect of code execution being considered. For *Line Executions (LE)*, we use the maximum number of executions recorded for each line across all execution traces. This choice reflects the assumption that these maximum values are more impactful on execution time. For the other code execution aspects, we select a single trace corresponding to a random test case to annotate the slow code.

C. Evaluation Metrics

To evaluate the model effectiveness in code optimization, we use three evaluation metrics introduced in prior work [10]:

- *Correct (%)*: The percentage of generated programs that successfully execute and produce the expected output for all available test cases.
- *Speedup*: The absolute improvement in terms of execution time. It is defined as $Speedup = \frac{T_I}{T_G}$, where T_I and T_G denote the execution times of the input and generated programs, respectively. In line with previous work [10], we set $Speedup = 1$ when the generated programs are either incorrect or slower than the input program.
- *Percent Optimized (%Opt)*: The percentage of generated code that is both correct and faster across the entire testing set. A generated program is considered faster if it achieves a speed improvement of at least 10%, corresponding to a *Speedup* of at least 1.1x.

D. Research Questions

This study aims to answer the following research questions:

- RQ₁** *How does learning line executions impact the effectiveness of language models for code optimization?* Several code optimizations aim to reduce the frequency of line executions (e.g., loop optimization [29]). This research question investigates the impact of teaching a model to understand line execution behavior on its ability to optimize code.
- RQ₂** *How does learning line coverage impact the effectiveness of language models for code optimization?* Here, we study how incorporating line coverage information into the language model affects its effectiveness in code optimization.
- RQ₃** *How does learning branch coverage impact the effectiveness of language models for code optimization?* Branch coverage information gives indications of how the conditional control flow statements impacts the code execution at run-time. This information can be crucial to identify potential optimization opportunities [30]. In this research question, we evaluate how learning such information influences the model effectiveness in optimizing code.
- RQ₄** *How does learning variable states impact the effectiveness of language models for code optimization?* Having an understanding of the types and values of variables may provide meaningful insights into code execution behaviors, e.g., variables elimination, type optimization and how the instructions reflects on the final values of used variables. Here we analyze whether teaching variable states to language models could enhance their ability to optimize code.

To answer these RQs, we conducted an empirical study to evaluate the effectiveness of twelve execution-aware language models for code optimization. For each code execution aspect (i.e., RQ), we developed three execution-aware models using the training strategies outlined in Section III-B. We then evaluated the effectiveness of these models using the metrics defined in Section III-C and compared to the standard CodeT5+ model.

IV. EXPERIMENTAL SETUP

In this section, we outline a detailed description of our experimental procedure, including the datasets construction, training procedure, baselines and models evaluation.

A. Datasets Construction

Strategies S_1 and S_2 require an execution-aware pre-training dataset as well as a fine-tuning dataset, whereas the strategy S_3 only needs an execution-aware fine-tuning dataset. We start by describing the data related to the code optimization task and execution information, followed by an explanation of how this data is transformed into datasets tailored to each specific training strategy.

1) *Code optimization*: We train language models for code optimization using the *Performance Improving Edits (PIE)* dataset [10], which consists of C++ slow-fast program pairs derived from from IBM *CodeNet* [31]. *CodeNet* is a large-scale dataset of programs, with each program representing a solution to a specific coding problem. It includes multiple programs for each coding problem, which may be authored either by the same developer or by different developers. Additionally, it provides test cases that can be used to evaluate the correctness of each program (based on the problem specifications) and measure its execution time. The *PIE* dataset involve 77k ‘slow-fast’ program pairs, where both programs in each pair aim to solve the same coding problem. *PIE* is already provided in a pre-split format consisting of train/validation/test subsets, ensuring that programs belonging to each problem only appear in one of those subsets. Additionally, we apply some preprocessing steps to canonicalize the code and remove any subjective elements in comments and coding style, as outlined in previous work [28]. We use the *gcc* preprocessor to remove comments and *clang-format* to format programs based on the LLVM coding style.

2) *Execution traces*: To obtain code execution information, we use the *gdb*² debugger to trace program execution, following a method similar to that used in [18]. To prevent prolonged computations, the tracing duration was limited to 500 seconds. The execution tracing procedure may fail if it takes too long or causes an invalid trace due to an unexpected execution interruption (e.g., program crash). The output of the tracing procedure consists of program execution traces that document the complete execution history of each program run. Specifically, such execution traces list all the executed lines, alongside with the name, type, and value of the corresponding program variables at each step of the program execution. We then preprocess these traces to extract execution information needed for our study, such as the execution frequency of each line of code and the state of variables at the end of program execution. For sake of clarity, we depict a sample execution trace referring to a dummy program in Figure 2. As illustrated, the trace displays the name, type and value of each variable for every line of code that is run, represented as blocks. The last

²<https://sourceware.org/gdb/>

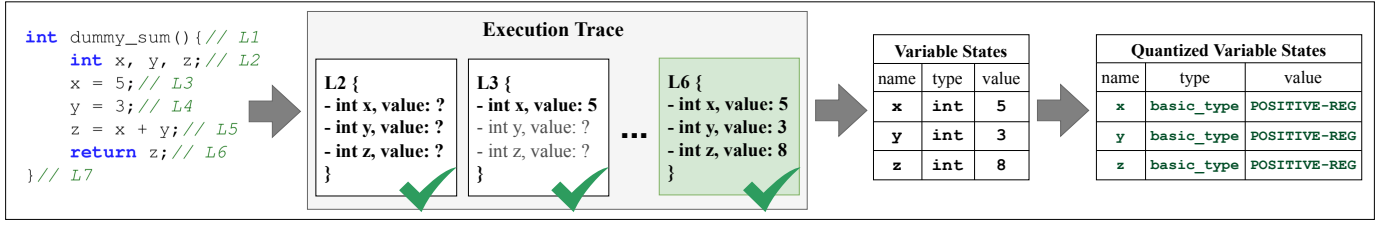


Fig. 2: The figure presents the execution trace corresponding to the outlined sample function, namely `dummy_sum()`. We use the question mark symbol (?) to show that a variable does not yet have an assigned value. Additionally, it also depicts the *variable states* information and the associated quantized values.

block in the trace (highlighted in green in the figure) contains the final variable information we use for the *variables state* execution aspect.

3) *Datasets*: For fine-tuning the model for code optimization, we directly rely on the *PIE* dataset for both the S_1 and S_2 strategies. Conversely, in execution-aware fine-tuning (i.e., S_3), the dataset must be adapted to incorporate execution information into the *slow* input program. To build the S_3 execution-aware fine-tuning dataset, we performed the tracing procedure over all the programs included in the ‘*slow*’ column of *PIE* dataset. Note that, a ‘*slow*’ program can appear multiple times in the *PIE* training dataset if associated to multiple faster versions. Specifically, among the 21,713 distinct ‘*slow*’ programs contained in the *PIE* training set, 9,145 programs were successfully traced. Similarly, from 978 ‘*slow*’ programs contained in the test set, 466 were traced successfully. This resulted in a total of 31,585 slow-fast training pairs and 466 instances for the testing set.

Differently, to construct the execution-aware pre-training dataset (for S_1 and S_2), we initially excluded all the C++ *CodeNet* programs appearing in either the ‘*slow*’ or ‘*fast*’ columns of the entire *PIE* dataset (43,606 programs). This filter prevents duplicate samples present in both the pre-training and fine-tuning datasets. Additionally, for each coding problem in *CodeNet*, we randomly select up to 150 program-test cases pairs, and we collect their corresponding execution traces. This approach ensures traces collection in a reasonable timeframe, as *CodeNet* comprises more than 8 millions of C++ programs. Indeed, the tracing collection procedure resulted in 119,764 execution traces for building the execution-aware pre-training dataset, involving 87,705 distinct programs. It is important to note that the same program may appear multiple times in this dataset if the corresponding coding problem involves multiples test cases, as each test case generates distinct execution traces.

B. Quantization Strategy

Language models may face challenges in learning code execution information, as they are primarily continuous numerical values [18]. Let us consider the example of line execution, where each line is associated with the number of times it has been executed for a specific input. This value can be any integer greater than or equal to zero. Thus, it may be challenging for the model to learn the underlying patterns behind this code execution aspect. To address this, as proposed in prior

work [18], we define a set of special tokens for each execution aspect to quantize the concrete execution information. For *line executions*, we performed a quantization of concrete values based on specific ranges, resulting in four unique tokens; $\langle e \rangle$: 1 execution, $\langle e+ \rangle$: 2–5 executions, $\langle E \rangle$: 6–20 executions, $\langle E+ \rangle$: 21+ executions. Lines that are not executed are not marked with any token. The majority of program lines were observed to have zero or one single execution, representing over 80% of all analyzed lines. In light of this, we allocated a dedicated token ($\langle e \rangle$) for representing a single execution case. We determined the other thresholds by studying how often lines of code were executed more than once. In particular, we set the higher threshold using the outlier detection rule $T = Q3 + 2.5 * IQR$, and subsequently divided the remaining values into two groups, using the median value as the dividing criterion. The calculated values for median and T were 4 and 23, which we rounded to 5 and 20. To identify the executed lines in the *line coverage* aspect, we only re-use the $\langle e \rangle$ token; uncovered lines remain unlabeled, as for the *LE* aspect. For learning *branch coverage* we use two tokens, namely $\langle BC \rangle$ and $\langle BNC \rangle$, to indicate that a line is part of a branch that is covered or not covered, respectively. In this case, lines that do not belong to any branch are left unlabeled. Regarding *variable states*, we adapt the quantized representation introduced in TRACED [18] to represent C++ variables, using their name, type and value. In Figure 2, we report the variable states quantization step for a dummy example. As depicted, the `int` became `basic_type`, while the concrete values have been converted into `POSITIVE-REG`. We outline all details related to the quantization strategy for each execution aspect in our replication package description.

In Table I, we provide a comprehensive example of the execution information that we feed into the language models. In the first column, we report the traced program alongside with a test case (i.e., program input) and the output obtained from its execution. The other columns outline the special tokens computed for each execution aspect. As shown in the table, the *variable states* information refers the entire code execution, whereas the other aspects correspond to line-by-line execution information.

C. Training Procedure

We exploit a Text-To-Text Transfer Transformer (T5) model, namely CodeT5+ [22], which is specifically pre-trained on

TABLE I: Sample C++ source code aligned with the corresponding execution information.

C++ Source Code	Line Exec.	Line Coverage	Branch Coverage	Variable States		
				Var. Name	Var. Type	Quantized Val.
Test Case: keyofscience						
#include<iostream>	-	-	-	k	class	OTHER
#include<string>	-	-	-	S	class	OTHER
using namespace std;	-	-	-	s	basic_type	POSITIVE-REG
int main() {	<e>	<e>	-	f	basic_type	POSITIVE-REG
int s=0, f=0;	<e>	<e>	-	i	basic_type	POSITIVE-REG
string S, k="keyence";	<e+>	<e>	-			
cin>>S;	<e>	<e>	-			
for(int i=0; i<S.length(); i++) {	<e+>	<e>	<BC>			
if (S[i]==k[i]) s++;	<e+>	<e>	<BC>			
else break;	<e>	<e>	<BC>			
}	-	-	-			
for(int i=0; i<S.length(); i++) {	<e+>	<e>	<BC>			
if (S[S.length()-1-i]==k[6-i]) f++;	<e+>	<e>	<BC>			
else break;	<e>	<e>	<BC>			
}	-	-	-			
if (s+f>=7) cout<<"YES"<<endl;	<e>	<e>	<BC>			
else cout<<"NO"<<endl;	-	-	<BNC>			
return 0;	<e>	<e>	-			
}	<e>	<e>	-			
Output: YES						

TABLE II: Overview of the total number of instances in the datasets after applying the maximum token limits filter. Note that different execution aspects can lead to dataset sizes, as the number of tokens may be different.

Dataset	BL	LE	LC	BC	VS
Execution-aware Pre-training	-	96,308	96,463	96,463	96,468
Fine-tuning	52,471		52,471		
Execution-aware Fine-tuning	28,753	28,654	28,244	28,654	28,034

source code and widely adopted in software engineering literature [32]–[35]. With regard to the available computational resources, we selected the CodeT5+ model with a parameter size of 220M. We ran all the experiments using the HuggingFace Transformers library. For all the tasks, we train the model for 1 epoch, with a batch size of 16 and a learning rate of $1e-5$. All experiments were conducted using the default AdamW optimizer. The fine-tuning script included in the official CodeT5+ release specifies the maximum token limits for the source and target inputs as 320 and 128, respectively³. Since the majority of our instances exceeds these threshold, we decided to increase both the source and target limits to 512 tokens. In Table II, we report the overall instances for each dataset resulting from filtering out the training samples with a number of tokens greater than 512.

In the following, we outline our implementation of each training strategy.

1) *Strategy S_1* : Concerning the S_1 execution-aware pre-training, the language model reads a program code and an associated test case (*i.e.*, program input) and generates the corresponding execution information. Let us define the set of the program lines as $L = \{l_1, \dots, l_n\}$ and

the test case as *input*; the model prompt is then built as $I_{PT_S_1} = \{classify: + input + \langle SEP \rangle + l_1, \dots, l_n\}$. The corresponding target sequence contains the execution information tokens joined together, separated by the “\n” character. Concerning line-wise execution information (*i.e.*, *LE*, *LC* and *BC*), we can define the set of execution information tokens corresponding to the program L as $E = \{e_1, \dots, e_n\}$ and accordingly build the target sequence as $O_{PT_S_1} = \{e_1, \dots, e_n\}$. For instance, assuming to deal with *line executions* data, the output sequence corresponding to the example reported in Table I would be encoded as “\n\n\n<e>\n<e>\n<e+>\n<e> . . . <e>\n<e>”. For the *variable states*, the target sequence consists of a list of inline code comments detailing the name, type, and value of each program variable. For instance, the code reported in Table I would generate five comments to append to the code, in the following way: “. . . // k class OTHER\n. . . \ni basic_type POSITIVE-REG”. $I_{PT_S_1}$ and $O_{PT_S_1}$ are fed into the language model as source and target sequences in a sequence-to-sequence supervised learning setting.

For the S_1 fine-tuning, the code optimization task is modeled as a sequence-to-sequence downstream tasks, in which the model reads the ‘slow’ code and generates the optimized version. Assuming to have a slow program composed of n lines ($S = \{s_1, \dots, s_n\}$) and a faster version composed of m lines ($F = \{f_1, \dots, f_m\}$), the input sequence for the fine-tuning task is formulated as $I_{FT_S_1} = \{optimize: + s_1, \dots, s_n\}$. The target sequence simply consists of the set of lines of the optimized version joined together: $O_{FT_S_1} = \{f_1, \dots, f_m\}$. The ‘classify:’ and ‘optimize:’ prefixes are utilized to assist the language model in distinguishing between the specified tasks.

2) *Strategy S_2* : This strategy is similar to strategy S_1 , but it integrates the *masked language modeling* (MLM) objective

³<https://github.com/salesforce/CodeT5/tree/main/CodeT5%2B>

to the execution-aware pre-training process. The model alternates the batches of the execution-aware pre-training dataset for performing both execution-aware and MLM pre-training objectives, aligned with prior research [36]. To distinguish between the two tasks, we use the ‘mlm:’ input prefix for MLM batches instances. Formally, let $M = \{m_1, \dots, m_k\}$ representing the set of k tokens encoding the input program, including those that are masked. The input sequence for MLM is then defined as $I_{PT-S_2} = \{mlm: + m_1, \dots, m_k\}$, with the output being the sequence of real tokens corresponding to the masked ones. The instances contained in the batches that are not used for MLM are handled identically to S_1 . With this feeding process, we aim to teach the language models to generate both execution and MLM labels. The fine-tuning stage is performed identically to S_1 , meaning that $I_{FT-S_2} = I_{FT-S_1}$ and $O_{FT-S_2} = O_{FT-S_1}$.

3) *Strategy S_3* : The strategy S_3 relies to a direct execution-aware fine-tuning process, in which the tokens related to execution information are directly incorporated within the *slow* code, by using code comments. For instance, the 8th line of code in Table I would be converted into the line `for(int i=0; i<S.length(); i++){ // <e+>` in case of *line executions*. For *line executions*, *line coverage* and *branch coverage* aspects, the execution information is generated for each line of code. Then, formalizing the set of source code lines annotated with executions tokens as $S_E = \{se_1, \dots, se_n\}$, the input sequence (I_{FT-S_3}) is constituted by the flattened version of S_E . Differently, for the *variable states* aspect we generate inline comments for each variable, similarly to S_1 , and append them to the ‘*slow*’ program. The target sequence O_{FT-S_3} corresponds to the optimized version of the code, as for the other strategies.

D. Baselines

We compare each execution-aware language model against the plain CodeT5+ model directly fine-tuned for the code optimization task (*i.e.*, without injecting any execution information).

In that, we define the following baseline models:

- *Baseline for S_1 - S_2 ($BL_{S_{12}}$)*: We train this baseline using the fine-tuning dataset outlined in Table II, involving 52,471 overall instances. Note that the dataset used for training the baseline is identical to the dataset used for fine-tuning the execution-aware models with S_1 and S_2 .
- *Baseline for S_3 (BL_{S_3})*: To guarantee a fair comparison regarding strategy S_3 , we restrict the analysis to the subset of programs that were successfully traced, applying the same subset for training the baseline, leading to a total of 28,753 instances.

E. Models Evaluation

We evaluate the effectiveness of each language model using the *PIE* test set [10]. Specifically, we prompt each model to generate an optimized version of each ‘*slow*’ program contained in the test set. Subsequently, the ‘*slow*’ programs and the generated programs are compiled and simulated using

the available *CodeNet* test cases on the *gem5* simulator [37]. Through this process, we assess the correctness of the generated programs and measure their execution times, following the methodology outlined in previous work [10]. The *Speedup* for an individual generated program is calculated by averaging the speedups obtained across all the test cases.

The total amount of time required for conducting all the experiments — including execution traces collection, model pre-training, fine-tuning and evaluation — is approximately 3 weeks. The traces collection and evaluation procedures were conducted on a linux server equipped with 40 Intel(R) Xeon(R) 2.30GHz CPUs and 78Gb of RAM, while the models training and inference have been performed over a CentOS HPC cluster equipped with 32 Intel(R) Xeon(R) Gold 6140M CPUs and two Nvidia A100 and A30 GPUs.

V. RESULTS

Our evaluation follows a common methodology across all the research questions. For each execution-aware and baseline model, we first generate the optimized code and then compute the evaluation metrics defined in Section III-C. Table III presents the results for the twelve execution-aware models and the two baselines. For the *Speedup* metric, we report the mean *Speedup* across all the programs of the test set. The first column of the table specifies the execution-aware or baseline model being evaluated. For instance, LE_{S_2} denotes the execution-aware model trained with line executions information (*LE*), using the strategy S_2 . $BL_{S_{12}}$ indicates the baseline model for strategies S_1 and S_2 , while BL_{S_3} denotes the baseline model for S_3 .

Additionally, we assess the statistical significance of the differences between the speedup provided by the execution-aware model and baseline model using Wilcoxon’s signed-rank test [38], and report two measures of effect size: Vargha-Delaney \hat{A}_{12} [39] and the matched pairs rank biserial correlation r [40]. In this context, the \hat{A}_{12} indicates the proportions of pairs where the *speedup* is higher than the baseline. An $\hat{A}_{12} > 0.5$ indicates that a particular execution-aware model generates more efficient source code than the corresponding baseline. The matched pairs rank biserial correlation r represents the difference between the proportion of favorable and unfavorable evidence. With regard to our work, favorable evidence is represented by a higher speedup for the execution-aware strategy, meaning that an $r > 0$ is denoting that the execution-aware model performs better than the baseline. Results from the comparison employing the Wilcoxon’s signed-rank test, along with effect sizes, are reported in Table IV. In the following, we discuss the achieved results by research questions.

A. *RQ₁: How does learning line executions impact the effectiveness of language models for code optimization?*

By observing Table III, we find that both LE_{S_1} and LE_{S_2} models perform generally worse than the $BL_{S_{12}}$ baseline across all the evaluation metrics. Concerning the correctness of the generated code, we observe lower percentages compared to the baseline, with 12.36% correct programs for LE_{S_1}

TABLE III: *Correctness, Speedup and %Opt* scores achieved by each baseline and proposed training strategy in performing code optimization of C++ programs. Scenarios that do not include pre-training have been reported in the bottom part of the table for a better understanding. For both the double-staged and single-staged strategies, the highest number in each column is **bolded** and the second-highest is underscored.

Model	Execution Aspect	Training Strategy		Evaluation Metrics		
		Pre-training	Fine-tuning	Correct	Speedup	%Opt
$BL_{S_{12}}$	-	-	Code optimization	18.75%	1.79	7.68%
LE_{S_1} LE_{S_2}	Line Executions	Execution-aware Execution-aware + MLM	Code optimization Code optimization	12.36% 11.97%	1.52 1.54	5.73% 5.6%
LC_{S_1} LC_{S_2}	Line Coverage	Execution-aware Execution-aware + MLM	Code optimization Code optimization	14.84% 14.45%	1.7 <u>1.76</u>	6.9% <u>7.55%</u>
BC_{S_1} BC_{S_2}	Branch Coverage	Execution-aware Execution-aware + MLM	Code optimization Code optimization	13.93% 13.15%	1.67 1.55	7.03% 5.73%
VS_{S_1} VS_{S_2}	Variable States	Execution-aware Execution-aware + MLM	Code optimization Code optimization	<u>15.49%</u> 14.97%	1.69 1.65	7.29% 6.64%
BL_{S_3}	-	-	Code optimization	<u>12.06%</u>	<u>2.09</u>	<u>9.22%</u>
LE_{S_3}	Line Executions	-	Execution-aware code optimization	12.71%	2.11	9.35%
LC_{S_3}	Line Coverage	-	Execution-aware code optimization	9.97%	1.67	5.84%
BC_{S_3}	Branch Coverage	-	Execution-aware code optimization	8.65%	1.64	5.76%
VS_{S_3}	Variable States	-	Execution-aware code optimization	11.55%	1.96	8.35%

TABLE IV: Comparison of speedups achieved by execution-aware models vs. baselines, evaluated using Wilcoxon test, Vargha-Delaney \hat{A}_{12} , and rank biserial correlation (r).

Exec-aware vs. Baseline	p -value	\hat{A}_{12}	r
LE_{S_1} vs. $BL_{S_{12}}$	<0.05	0.446 (N)	-0.643
LE_{S_2} vs. $BL_{S_{12}}$	<0.05	0.467 (N)	-0.399
LE_{S_3} vs. BL_{S_3}	0.064	-	-
LC_{S_1} vs. $BL_{S_{12}}$	<0.05	0.451 (N)	-0.633
LC_{S_2} vs. $BL_{S_{12}}$	<0.05	0.455 (N)	-0.515
LC_{S_3} vs. BL_{S_3}	<0.05	0.471 (N)	-0.416
BC_{S_1} vs. $BL_{S_{12}}$	<0.05	0.452 (N)	-0.536
BC_{S_2} vs. $BL_{S_{12}}$	<0.05	0.447 (N)	-0.61
BC_{S_3} vs. BL_{S_3}	<0.05	0.47 (N)	-0.503
VS_{S_1} vs. $BL_{S_{12}}$	0.069	-	-
VS_{S_2} vs. $BL_{S_{12}}$	0.075	-	-
VS_{S_3} vs. BL_{S_3}	<0.05	0.487 (N)	-0.449

and 11.97% for LE_{S_2} , versus a correctness of 18.75% for $BL_{S_{12}}$. We also observe that execution-aware pre-training works slightly better for correctness when combined with masked language modeling (*i.e.*, LE_{S_2}). We find that the speedup achieved by the code generated by execution-aware models (1.52x and 1.54x for LE_{S_1} and LE_{S_2} , respectively) is not as good as the one reached by the baseline (1.79x). The %Opt values confirm the same trend observed for the other two metrics, with lower percentages of optimized programs for the execution-aware models.

Contrariwise, we find that execution-aware fine-tuning (LE_{S_3}) works slightly better than its baseline BL_{S_3} across all the evaluation metrics. Particularly, LE_{S_3} reports a higher number of correct programs (12.71% vs. 12.06%), a highest speedup (2.11x vs. 2.09x), and a higher percentage of

optimized programs (9.35% vs. 9.22). Since these improvements appear marginal, we further investigated the statistical significance of difference between the speedups provided by LE_{S_3} versus those of BL_{S_3} . Table IV shows the results of the Wilcoxon test, which suggest no statistical difference for the speedup, *i.e.*, $p > 0.05$.

Answer to RQ₁: Teaching line execution behavior to the language model does not enhance its code optimization capabilities. Execution-aware fine-tuning shows a slight improvement, but without statistical significance.

B. RQ₂: How does learning line coverage impact the effectiveness of language models for code optimization?

Results for line coverage reveal a trend similar to that observed in RQ₁, with baselines outperforming all three execution-aware models. When comparing LC_{S_1} and LC_{S_2} , we observe an increase in speedup and %Opt values when the MLM is combined with execution-aware pre-training (1.76x vs. 1.7x for speedup, and 7.55x vs. 6.9x for %Opt), despite a slight reduction in correctness (14.45% vs. 14.84%).

By observing Table III, we notice that, compared to other execution-aware models based on the strategies S_1 and S_2 , line coverage demonstrates the highest effectiveness in terms of speedup and the percentage of optimized programs. This underlines that the effectiveness of execution-aware training strategies are closely related to the specific execution aspect used to feed the model. For the execution-aware fine-tuning strategy LC_{S_3} , we observe a similar trend, showing lower effectiveness compared to the baseline BL_{S_3} , *e.g.*, speedup of 1.67x vs. 2.11x.

Wilcoxon's test indicates statistically significant differences for all comparisons between execution-aware models and

baselines ($p < 0.05$), with a negligible disadvantage for all execution-aware models in terms of effect size ($\hat{A}_{12} < 0.5$).

Answer to RQ₂: Learning line coverage does not improve the model effectiveness in optimizing code. Execution-aware models perform worse than baseline models across all metrics.

C. RQ₃: How does learning branch coverage impact the effectiveness of language models for code optimization?

As shown in Table III, execution-aware models consistently underperform compared to their respective baselines across all evaluation metrics. Specifically, we observe reduced correctness, with values of 13.93%, 13.15%, and 8.65% for BC_{S_1} , BC_{S_2} , and BC_{S_3} , respectively. Regarding optimization metrics, execution-aware models also lag behind the baselines, achieving speed-ups of up to 1.67x (compared to 1.79x for $BL_{S_{12}}$ and 2.09x for BL_{S_3}) and %Opt values of 7.03% (vs. 7.68% for $BL_{S_{12}}$ and 9.22% for BL_{S_3}).

When focusing exclusively on execution-aware fine-tuning, BC_{S_3} exhibits worse effectiveness across all evaluation metrics compared to the baseline BL_{S_3} .

Answer to RQ₃: The integration of branch coverage information into language models reduces both correctness and optimization metrics across the execution-aware training strategies.

D. RQ₄: How does learning variable states impact the effectiveness of language models for code optimization?

The results for variable states in Table III indicate a degradation in all evaluation metrics when employing any of the considered execution-aware strategies. Specifically, correctness decreases on average from 18.75% for the baseline $BL_{S_{12}}$ to 15.49% for the execution-aware model VS_{S_1} and further to 14.97% for the VS_{S_2} model. Regarding execution-aware fine-tuning, we observe a similar decline in correctness, from 12.06% ($BL_{S_{12}}$) to 11.55% (VS_{S_3}). Speedup and %Opt also follow a similar downward trend compared to the baselines.

Answer to RQ₄: Learning variable states in language models does not enhance the effectiveness of code optimization. On the contrary, we observe reduced correctness and fewer optimizations in all execution-aware models.

VI. DISCUSSION

We conduct a complementary analysis to better understand the results of our study and to identify potential insights for future research on execution-aware code optimization.

A. Correctness analysis

Our evaluation reveals the limited capability of execution-aware models to produce correct code, *i.e.*, code that successfully passes all test cases. To better understand this limitation, we delve deeper into the reasons behind the lack of correctness. In particular, we analyze the proportions of generated programs that: (i) exhibit well-formed syntax (*i.e.*, the code

TABLE V: Percentages of programs that successfully compile, execute, and produce correct outputs.

Model	Compiled	Executed	Correct
$BL_{S_{12}}$	78.65%	76.95%	18.75%
LE_{S_1}	81.25%	78.65%	12.36%
LE_{S_2}	80.47%	77.21%	11.97%
LC_{S_1}	85.68%	82.94%	14.84%
LC_{S_2}	74.48%	72.53%	14.45%
BC_{S_1}	80.08%	77.08%	13.93%
BC_{S_2}	89.19%	86.72%	13.15%
VS_{S_1}	85.03%	82.29%	15.49%
VS_{S_2}	85.68%	80.99%	14.97%
BL_{S_3}	89.83%	88.65%	12.06%
LE_{S_3}	86.81%	84.65%	12.71%
LC_{S_3}	86.13%	84.18%	9.97%
BC_{S_3}	85.10%	84.38%	8.65%
VS_{S_3}	86.73%	84.77%	11.55%

successfully compiles); (ii) do not result in runtime errors (*i.e.*, the code successfully executes); and (iii) produce the expected output (*i.e.*, the code passes all test cases)

Table V presents the percentages of programs that successfully compile (*Compiled*), execute without errors (*Executed*), and produce the expected output (*Correct*). We observe that execution-aware models exhibit comparable, and sometimes superior, effectiveness in generating programs that successfully compile and execute. Specifically, when considering execution-aware pre-training strategies (S_1 and S_2), these models outperform the baseline $BL_{S_{12}}$ in generating programs that compile and execute successfully. For example, LE_{S_1} , LC_{S_1} , BC_{S_2} , and VS_{S_2} generate programs that successfully compile in 81.25%, 85.68%, 89.19%, and 85.68% of cases, respectively, compared to 78.65% for $BL_{S_{12}}$. Similarly, LE_{S_1} , LC_{S_1} , BC_{S_2} , and VS_{S_1} generate programs that successfully execute in 78.65%, 82.94%, 86.72%, and 82.29% of cases, respectively, compared to 76.95% for $BL_{S_{12}}$. However, when considering the correctness of the generated programs, we observe a notable drop in effectiveness for execution-aware models. Models using pre-training strategies S_1 and S_2 achieve a correctness of up to 15.49%, whereas the baseline $BL_{S_{12}}$ achieves a correctness of 18.75%. These results suggest that while execution-aware models generally perform well in generating well-formed code that successfully executes, they may be limited in their semantic understanding of the intended functionality (*i.e.*, correctness). This finding is somewhat counterintuitive, as execution-aware training strategies are typically designed to enhance language models' semantic understanding of code [15], [16], [18]. Nevertheless, in light of these findings, we encourage future research on code optimization to focus on improving the correctness of programs generated by execution-aware language models. This could involve exploring additional aspects of code execution or developing alternative, more effective training strategies.

TABLE VI: Speedup statistics considering only (i) correct programs and (ii) optimized programs.

Model	Speedup			
	Correct Programs		Optimized Programs	
	Instances	Mean	Instances	Mean
$BL_{S_{12}}$	144	5.2x	141	5.29x
LE_{S_1}	95	5.2x	91	5.39x
LC_{S_1}	114	5.71x	60	9.96x
BC_{S_1}	107	5.83x	63	9.21x
VS_{S_1}	119	5.44x	111	5.76x
LE_{S_2}	92	5.49x	90	5.58x
LC_{S_2}	111	6.22x	66	9.79x
BC_{S_2}	101	5.21x	55	8.72x
VS_{S_2}	115	5.37x	109	5.61x
BL_{S_3}	51	10.12x	50	10.3x
LE_{S_3}	53	9.76x	47	10.87x
LC_{S_3}	41	7.76x	26	11.67x
BC_{S_3}	36	8.4x	26	11.24x
VS_{S_3}	47	9.29x	40	10.74x

B. Speedup analysis

We assess the average speedup across the entire set of generated code, accounting for both incorrect and slower programs. In line with prior research [10], we assign a *speedup* = 1 in such cases. To better understand the speedup provided by execution-aware models, here we focus on analyzing only the correct programs generated by the models, as well as the programs that achieve a non-negligible optimization of execution time (greater than 1%). Table VI presents the mean speedup achieved by each model, calculated exclusively for programs that produce the expected outputs (*Correct Programs*) and for those that optimize the input program (*Optimized Programs*). Additionally, we report the number of program instances considered in each case. Execution-aware models deliver comparable or even superior speedup compared to baseline models when considering only correct programs. This trend is more evident in models utilizing execution-aware pre-training (*i.e.*, S_1 and S_2): BC_{S_1} and LC_{S_2} achieve mean speedups of 5.83x and 6.22x, respectively, whereas the baseline $BC_{S_{12}}$ achieves a speedup of 5.2x. Furthermore, when focusing on optimized programs, we observe that execution-aware models achieve significantly higher speedups compared to baseline models. For instance, LC_{S_1} achieves a speedup of 9.96x, while the baseline $BL_{S_{12}}$ 5.29x. Similarly, LC_{S_3} achieves a speedup of 11.67x, compared to 10.3x for the baseline BL_{S_3} . These findings indicate that execution-aware models have the potential to deliver significant performance improvements when they successfully generate correct or optimized programs.

C. Threats to Validity

1) *Construct validity*: Considering different execution aspects and quantization criteria could produce results that differ from those presented in this study. Nevertheless, we focused on execution aspects that have been successfully employed in prior work [18]. Using alternative training strategies to

learn code execution information or larger language models may yield different outcomes. Still, we evaluated a total of twelve execution-aware models by employing three distinct training strategies, and four execution aspects. Additionally, we employed CodeT5+, a language model widely adopted in the software engineering literature [32]–[35].

2) *Internal validity*: The choice of the language model hyperparameters plays a key role in determining the model effectiveness. We try to mitigate possible biases by adopting the default parameters of the CodeT5+ release package. However, choosing different hyperparameters may lead to different results. Measuring program execution time typically involves a certain degree of variability [41]. To mitigate this risk and improve the scalability of our experiments, similar to previous work [10], we rely on a deterministic simulator, namely *gem5*, to measure program execution time. However, executing programs on concrete hardware environments may yield different results. In addition, the reliability of our performance benchmarking is significantly affected by the choice of the test cases, which we directly inherit from CodeNet.

3) *External validity*: Our evaluation is based on C++ programs from CodeNet, which primarily consist of self-contained programs to solve specific coding problems. In addition, a random sampling was considered during dataset construction. The findings of this study, they may not generalize to programs written in other programming languages or to more complex software systems involving multiple modules, third-party libraries, or intricate interdependencies.

VII. CONCLUSION

We investigate the effectiveness of execution-aware language models in code optimization. We evaluate four execution aspects and three training strategies to teach a CodeT5+ how code executes at runtime. Our findings reveal that learning code execution behavior does not enhance the model’s ability to optimize code. A significant weakness of execution-aware models is their limited capacity to generate semantically correct code, even though they often produce syntactically correct programs. We encourage future research to explore additional code execution aspects, leverage other (larger) language models, and experiment with alternative training strategies. This research objective informs and drives our future agenda.

ACKNOWLEDGEMENTS

The authors from University of L’Aquila are partially supported by Italian Government (Ministero dell’Università e della Ricerca, PRIN 2022 PNRR): “RECHARGE: monitoring, tEsting, and CHAracterization of performAnce Regressions” (cod.P2022SELA7), and by “ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing”, funded by European Union – NextGenerationEU. Bavota acknowledges the support from the Swiss National Science Foundation (SNSF), project “PARSED” (grant agreement No. 219294).

REFERENCES

- [1] K. Eaton. (2012) How one second could cost amazon \$1.6 billion in sales. [Online]. Available: <https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales>
- [2] A. Aleti, B. Buhnova, L. Grunske, A. Koziolok, and I. Meedeniya, “Software architecture optimization methods: A systematic literature review,” *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.
- [3] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “A survey on compiler autotuning using machine learning,” *ACM Comput. Surv.*, vol. 51, no. 5, Sep. 2018. [Online]. Available: <https://doi.org/10.1145/3197978>
- [4] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024, just Accepted. [Online]. Available: <https://doi.org/10.1145/3695988>
- [5] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, “Large language models for software engineering: Survey and open problems,” in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023, pp. 31–53.
- [6] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, “Source Code Summarization in the Era of Large Language Models,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 419–431. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00034>
- [7] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, “Vulnerability Detection with Code Language Models: How Far Are We?,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 469–481. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00038>
- [8] Y. Xiao, V.-H. Le, and H. Zhang, “Demonstration-free: Towards more practical log parsing with large language models,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 153–165. [Online]. Available: <https://doi.org/10.1145/3691620.3694994>
- [9] L. Traini, F. Di Menna, and V. Cortellesa, “Ai-driven java performance testing: Balancing result quality with testing time,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 443–454. [Online]. Available: <https://doi.org/10.1145/3691620.3695017>
- [10] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. R. Gardner, Y. Yang, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, “Learning performance-improving code edits,” in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=ix7rLVHXYy>
- [11] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 254–266. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00021>
- [12] S. Garg, R. Z. Moghaddam, and N. Sundaresan, “Rapgen: An approach for fixing code inefficiencies in zero-shot,” *CoRR*, vol. abs/2306.17077, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.17077>
- [13] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, “Deepdev-perf: a deep learning-based approach for improving software performance,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 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>
- [14] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu, “Lms: Understanding code syntax and semantics for code analysis,” 2024. [Online]. Available: <https://arxiv.org/abs/2305.12138>
- [15] A. Ni, M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin, “Next: Teaching large language models to reason about code execution,” in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21–27, 2024*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=B1W712hMBi>
- [16] Y. Ding, J. Peng, M. J. Min, G. E. Kaiser, J. Yang, and B. Ray, “Semcoder: Training code language models with comprehensive semantics,” *CoRR*, vol. abs/2406.01006, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2406.01006>
- [17] J. Huang, J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen, “Code representation pre-training with complements from program executions,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, F. Dernoncourt, D. Preotiu-Pietro, and A. Shimorina, Eds. Miami, Florida, US: Association for Computational Linguistics, Nov. 2024, pp. 267–278. [Online]. Available: <https://aclanthology.org/2024.emnlp-industry.21>
- [18] Y. Ding, B. Steenhoeck, K. Pei, G. Kaiser, W. Le, and B. Ray, “Traced: Execution-aware pre-training for source code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3608140>
- [19] B. Souza and M. Pradel, “Lexecutor: Learning-guided execution,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1522–1534. [Online]. Available: <https://doi.org/10.1145/3611643.3616254>
- [20] T. Mammadov, D. Klakow, A. Koller, and A. Zeller, “Learning program behavioral models from synthesized input-output pairs,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.08597>
- [21] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, “Reasoning Runtime Behavior of a Program with LLM: How Far Are We?,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 140–152. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00012>
- [22] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, “Codet5+: Open code large language models for code understanding and generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6–10, 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Association for Computational Linguistics, 2023, pp. 1069–1088. [Online]. Available: <https://doi.org/10.18653/v1/2023.emnlp-main.68>
- [23] A. Mastropaolo, M. Ciniselli, L. Pascarella, R. Tufano, E. Aghajani, and G. Bavota, “Towards summarizing code snippets using pre-trained transformers,” in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15–16, 2024*, I. Steinmacher, M. Linares-Vásquez, K. P. Moran, and O. Baysal, Eds. ACM, 2024, pp. 1–12. [Online]. Available: <https://doi.org/10.1145/3643916.3644400>
- [24] A. Mastropaolo, F. Zampetti, G. Bavota, and M. D. Penta, “Toward automatically completing github workflows,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 2024, pp. 13:1–13:12. [Online]. Available: <https://doi.org/10.1145/3597503.3623351>
- [25] A. Seifia, S. Das, S. Shafiq, and N. Medvidovic, “Toward improved deep learning-based vulnerability detection,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 2024, pp. 62:1–62:12. [Online]. Available: <https://doi.org/10.1145/3597503.3608141>
- [26] T. Huang, Z. Sun, Z. Jin, G. Li, and C. Lyu, “Knowledge-aware code generation with large language models,” in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15–16, 2024*, I. Steinmacher, M. Linares-Vásquez, K. P. Moran, and O. Baysal, Eds. ACM, 2024, pp. 52–63. [Online]. Available: <https://doi.org/10.1145/3643916.3644418>
- [27] A. D. Gotmare, J. Li, S. Joty, and S. C. H. Hoi, “Efficient text-to-code retrieval with cascaded fast and slow transformer models,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 388–400. [Online]. Available: <https://doi.org/10.1145/3611643.3616369>
- [28] Z. Chen, S. Fang, and M. Monperrus, “Supersonic: Learning to generate source code optimizations in c/c++,” *IEEE Transactions on Software Engineering*, pp. 1–17, 2024.

- [29] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 370–380.
- [30] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 902–912.
- [31] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. R. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html>
- [32] T. Ahmed, K. S. Pai, P. T. Devanbu, and E. T. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 220:1–220:13. [Online]. Available: <https://doi.org/10.1145/3597503.3639183>
- [33] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=UnUwSIgK5W>
- [34] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. 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, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 37:1–37:12. [Online]. Available: <https://doi.org/10.1145/3597503.3623316>
- [35] A. Mastropaolo, V. Nardone, G. Bavota, and M. D. Penta, "How the training procedure impacts the performance of deep learning-based vulnerability patching," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024*. ACM, 2024, pp. 150–159. [Online]. Available: <https://doi.org/10.1145/3661167.3661200>
- [36] J. Huang, J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen, "Code representation pre-training with complements from program executions," *CoRR*, vol. abs/2309.09980, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2309.09980>
- [37] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. [Online]. Available: <https://doi.org/10.1145/2024716.2024718>
- [38] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: <http://www.jstor.org/stable/3001968>
- [39] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. [Online]. Available: <https://doi.org/10.3102/10769986025002101>
- [40] D. S. Kerby, "The simple difference formula: An approach to teaching nonparametric correlation," *Comprehensive Psychology*, vol. 3, p. 11.IT.3.1, 2014. [Online]. Available: <https://doi.org/10.2466/11.IT.3.1>
- [41] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 409–425. [Online]. Available: <https://www.usenix.org/conference/osdi18/presentation/maricq>