Calico: Automated Knowledge Calibration and Diagnosis for Elevating AI Mastery in Code Tasks

Yuxin Qiu
University of California at Riverside
Riverside, USA
yuxin.qiu@email.ucr.edu

Jie Hu
University of California at Riverside
Riverside, USA
jhu066@ucr.edu

Qian Zhang
University of California at Riverside
Riverside, USA
qzhang@cs.ucr.edu

Heng Yin
University of California at Riverside
Riverside, USA
heng@cs.ucr.edu

Abstract

Recent advancements in large language models (LLMs) have exhibited promising capabilities in addressing various tasks such as defect detection and program repair. Despite their prevalence, LLMs still face limitations in effectively handling these tasks. Common strategies to adapt them and improve their performance for specific tasks involve fine-tuning models based on user data or employing in-context learning with examples of desired inputs and outputs. However, they pose challenges for practical adoption due to the need for extensive computational resources, high-quality data, and continuous maintenance. Furthermore, neither strategy can explain or reason about the deficiencies of LLMs in the given tasks.

We propose Calico to address the high cost of fine-tuning, eliminate the necessity for task-specific examples, and provide explanations of LLM deficiency. At the heart of Calico is an evolutionary approach that interleaves knowledge calibration and AI deficiency diagnosis. The key essence of Calico is as follows. First, it focuses on identifying knowledge gaps in LLMs' program comprehension. Second, it conducts automated code refactoring to integrate the overlooked knowledge into the source code for mitigating those gaps. Third, it employs *what-if* analysis and counterfactual reasoning to determine a minimum set of overlooked knowledge necessary to improve the performance of LLMs in code tasks.

We have extensively evaluated Calico over 8,938 programs on three most commonly seen code tasks. Our experimental results show that vanilla ChatGPT cannot fully understand code structures. With knowledge calibration, Calico improves it by 20% and exhibits comparable proficiency compared to fine-tuned LLMs. Deficiency diagnosis contributes to 8% reduction in program sizes while ensuring performance. These impressive results demonstrate the feasibility of utilizing a vanilla LLM for automated software engineering (SE) tasks, thereby avoiding the high computational costs associated with a fine-tuned model.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680399

CCS Concepts

• **Software and its engineering → Software development techniques**; **Software testing and debugging**; **Dynamic analysis**; **Search-based software engineering**;

Keywords

Software engineering, software testing, large language model

ACM Reference Format:
Yuxin Qiu, Jie Hu, Qian Zhang, and Heng Yin. 2024. Calico: Automated Knowledge Calibration and Diagnosis for Elevating AI Mastery in Code Tasks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680399

1 Introduction

Large language models (LLMs) have gained substantial traction in software engineering (SE). These models, with their advanced comprehension and generation capabilities, are stimulating innovations in various areas of SE research and practices, such as defect detection [14, 25, 36, 57], clone detection [18, 78], code summarization [42, 51], and program repair [17, 71]. For example, Copilot [7] augments software developers' productivity by providing code suggestions derived from natural language descriptions and program context such as comments, function names, and the surrounding code.

Challenges and Current Practices. Despite their prevalence, recent studies have found that LLMs' effectiveness in handling SE tasks is limited. For example, the state-of-the-art (SOTA) LLM CoTexT records an accuracy of 65.99% in defect detection [51]. In bug fixing and code summarization tasks, the CodeT5 model correctly completes the tasks with accuracies of only 17.79% and 19.55%, respectively [51].

To enhance LLM performance in SE tasks, common strategies include: (a) *fine-tuning pre-trained models* with task-specific, user-annotated data [11, 24]. For example, alongside the general linguistic and commonsense knowledge acquired during pre-training, Huang et al. [27] fine-tune CodeBERT, GraphCodeBERT, PLBART, CodeT5, and UniXcoder to make them concentrate on automated program repair; and (b) utilizing *in-context learning with a few relevant examples* such as illustrative questions and corresponding

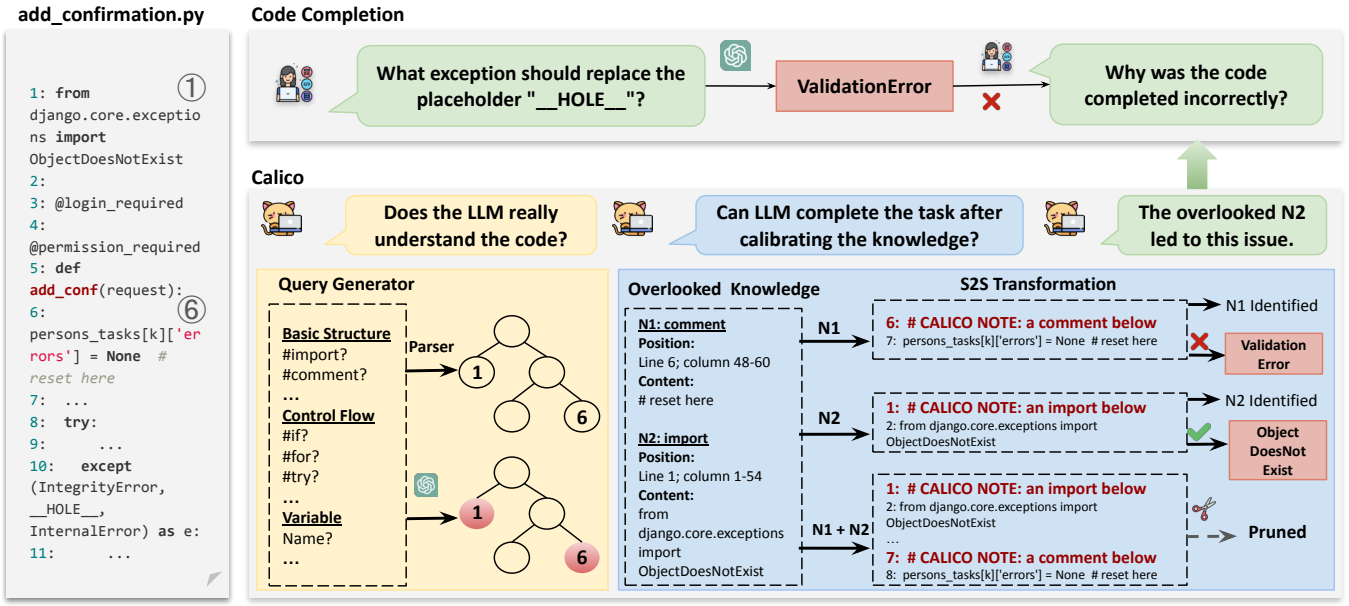


Figure 1: Working Example.

answers to enrich contextual comprehension for the target tasks [10, 14, 67, 69, 71]. For example, the protocol fuzzer ChatAFL [44] prompts ChatGPT to generate message grammar of internet protocols by providing two grammar examples of different protocols in the required format. While both fine-tuning and in-context learning paradigms enhance LLM performance, their practical application to code tasks in SE presents challenges. Such challenges typically arise from the following sources:

- *Computational Resources.* Fine-tuning inevitably requires a substantial amount of computational effort. It can be costly and sometimes infeasible for SOTA models with numerous parameters. Additionally, maintaining the relevance of fine-tuned models over time is difficult as new knowledge emerges, which often requires continuous updates with fresh data.
- *Data Availability and Similarity.* Both fine-tuning and in-context learning require high-quality, task-specific data, which may not always be readily available. Collecting such data often requires significant effort such as user annotations.
- *Lack of Explanations.* Both approaches work as a black box in code tasks. They do not offer explanations for the deficiencies of LLMs in certain scenarios, which creates a gap in our understanding of models' limitations and the strategies needed to overcome them.
- *Model Generalizability.* Both paradigms risk overfitting the LLM to specific examples or tasks, potentially reducing the model's ability to generalize to a broader range of SE scenarios.

Calico. We propose to *lay the groundwork* for enhancing the performance of LLMs in code tasks *without fine-tuning*. Specifically, we propose Calico, an evolutionary approach that interleaves knowledge **cali**bration and AI deficiency diagnosis for **co**de.

Indeed, the essence of fine-tuning or in-context learning with a few examples is to enhance an LLM's ability to understand code. We thus *hypothesize* that a similar improvement can be achieved by identifying fundamental knowledge gaps, such as overlooked code syntax, within the LLM's current understanding and then integrating the necessary information to fill those gaps.

The overall insight of Calico lies in its innovative *dual capability* to enhance LLM performance and provide explanations of task-specific deficiencies at the same time. Unlike in-context learning, which is limited to question-answer examples within a specific context to guide the LLM, Calico identifies and fills fundamental knowledge gaps across a variety of contexts. Thus, we can improve LLM performance in a broader range of tasks without the need for extensive task-specific fine-tuning. Calico also aims for the minimal integration of knowledge necessary to produce correct LLM responses, thereby addressing the explanations for deficiencies. Our key approach is three-fold as elaborated below.

1. AI Mastery Screening. Conventional in-context learning enhances LLM performance by supplementing task-relevant examples, while fine-tuning improves LLM effectiveness by feeding task-specific data. Based on the insight that a profound understanding of code structure is a must for effectively executing any code tasks, Calico initiates its process by examining how well LLMs understand source code. It first formulates queries that probe the structural comprehension of code. Next, it contrasts LLM's outputs with the ground truth from *treesitter* [1], an abstract syntax tree (AST) parser. This comparison allows Calico to identify the specific AST nodes and structures where LLMs overlook critical knowledge, thus shedding light on their limitations in code comprehension and offering a pathway to targeted improvements.

Figure 1 shows a code completion example using ChatGPT 3.5 [2]. When tasked with replacing the placeholder `__HOLE__` (Line 10) in `add_confirmation.py`, ChatGPT incorrectly suggests `ValidationError` instead of the correct `ObjectDoesNotExist` exception. To

improve precision, Calico first evaluates ChatGPT's code understanding by comparing its responses to queries about the AST against the ground-truth results from the Python AST parser in treesitter. The query generator in Calico generates these queries, which can involve the identification of library import, control flows, variable definitions, *etc.* This comparison uncovers ChatGPT's oversight of crucial AST nodes, `N1:comment` corresponding to the inline comment in Line 6 and `N2:import` corresponding to the library import in Line 1, and their associated information. This indicates the knowledge gaps `{N1, N2}` that need to be addressed.

2. Knowledge Calibration. Once Calico identifies the knowledge gaps, it integrates this information into the source code to *refocus* the LLM's attention on these specific areas. Such an integration is achieved through AST-based source-to-source code transformation. Calico inserts a comment node before each AST node associated with overlooked knowledge. In other words, it adds an annotation highlighting the subsequent line of code as needing extra attention. For example, in Figure 1, to refocus ChatGPT's attention to the inline comment in Line 6, Calico adds a comment `CALICO NOTE: a comment below` prior to the original line of comment.

3. Deficiency Diagnosis. Intuitively adding all overlooked knowledge to LLMs can overload the model with task-irrelevant data and does not offer insights into the model's task-specific deficiency. Thus, it is important to selectively integrate key information needed to keep LLMs focused and effective. There have been developments of counterfactual reasoning in AI [12] to find the smallest changes needed to alter a model's prediction. Calico utilizes such an approach through *what-if analysis* to investigate how filling different knowledge gaps affects LLM responses. By doing so, we can ensure that prompts remain concise with only the most significant knowledge, which can concurrently serve as an explanation for performance deficiencies. In fact, Calico aims to answer "What would happen to LLM responses if we were to fill alternative knowledge gaps?" To achieve this, Calico systematically generates all combinations of overlooked knowledge. Next, it traverses them from the smallest to the largest combination. When the smallest effective combination is found that accomplishes the task, it indicates the essential knowledge has been pinpointed and identified, and Calico concludes its process and explains the deficiency as lacking such identified knowledge.

In Figure 1, Calico identifies the overlooked knowledge `{N1:comment, N2:import}` and their three possible combinations—`C1 = {N1}`, `C2 = {N2}`, and `C3 = {N1, N2}`. It then systematically integrates each combination into the source code via knowledge calibration and subsequently prompts ChatGPT to re-evaluate its code completion. Calico starts from the smallest set `{N1}` and observes that emphasizing this inline comment does not yield the desired response because ChatGPT still outputs the wrong exception type `ValidationError`. Recognizing this persistence of incorrectness, Calico moves on to evaluate the next set `{N2}` to refocus ChatGPT's attention to the associated library import. Calico observes that ChatGPT accurately identifies the appropriate exception type, `ObjectDoesNotExist`, to replace the placeholder `__HOLE__`. This indicates that ChatGPT initially overlooked this library import, which led to the incorrect code completion. Calico identifies this minimal knowledge as essential for enhancing ChatGPT's ability to fulfill the code completion task effectively.

Results. We conduct an extensive evaluation over 8,938 programs on three commonly seen code tasks—bug detection, code summarization, and program repair. These programs are from five open-source, widely-used datasets, including Bugs2Fix [60], CodeSearchNet [28], Devign [77], ETHPy150Open [30], and InferredBugs [29]. They span *four* most commonly used programming languages—C, Java, JavaScript, and Python.

Our experimental results demonstrate that the vanilla ChatGPT understands 90% basic structures and API calls, and 80% control flow, variables, and expressions. With knowledge calibration, Calico improves the vanilla LLM by up to 20% and exhibits comparable performance with fine-tuned LLMs. Using counterfactual reasoning to integrate a minimum set of knowledge, Calico decreases the program size by an average of 8% compared to adding all overlooked knowledge while ensuring LLM performance. Per the open science policy, we make Calico's artifacts, benchmark programs, and datasets available at https://zenodo.org/records/13145331.

In summary, this work makes the following contributions.

- Calico presents a novel, lightweight, broadly accessible solution to enhance vanilla LLMs across various tasks without the high computational costs of fine-tuning.
- Our extensive study on 8,938 programs assesses LLMs' code structure comprehension, revealing diverse proficiency across programming languages.
- Calico examines LLMs' understanding of code and integrates the overlooked knowledge into the original code.
- Calico offers the first automated LLM diagnosis framework that identifies key information to keep LLMs focused and effective.

The rising demand for AI-enabled tools in the era of generative AI highlights the importance of integrating LLMs in software development effectively. To our knowledge, Calico stands out as a pioneering approach that can reshape the utilization of LLMs in SE by diminishing the necessity for fine-tuning. It works by pinpointing knowledge gaps, supplementing this knowledge within the current task, and employing counterfactual reasoning to explain performance deficiencies. While Calico initially targets gaps regarding code structure understanding, its approach can be generalized to other code-related knowledge, such as dataflow analysis, potentially revolutionizing a broad range of software analysis practices.

2 Background

2.1 Code Tasks in Software Engineering

Prior work [50, 51] summarizes the key code tasks in SE, which are classified as understanding tasks and generation tasks. Understanding tasks refer to activities of comprehending source code and include defect detection, clone detection, code search, *etc.* They involve a variety of input-output types. For example, in defect detection where the objective is to identify if the provided code contains any defects, the input is the source code, and the output yields a specific predicted value such as `True` or `False`.

Generation tasks refer to code-related translations and include code translation, bug fixing, code summarization, code generation, *etc.* Similarly, they involve various input-output types. In bug fixing

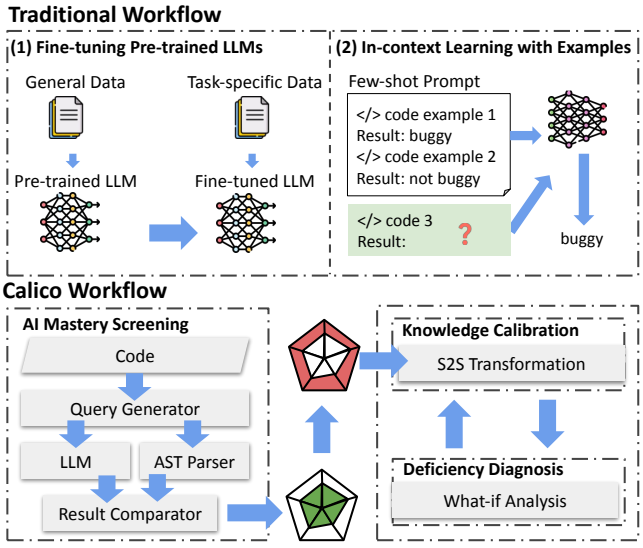


Figure 2: Calico Overview.

tasks for repairing buggy code by generating the correct version, both the input and the output are source code.

2.2 Large Language Models for Code Tasks

LLMs are extensively applied in the field of natural language processing [46, 64]. In SE, recent work [16, 19, 24, 27, 71, 72] have explored how to use LLMs for code tasks. For example, FuzzGPT [16] is a fuzzer to detect bugs in deep learning (DL) libraries. It can be either fine-tuned with additional programs for triggering similar bugs or prompted with a few context-aware examples.

LLMs offer the potential for fine-tuning task-specific models; however, the significant hardware resources and data required can be prohibitive for those without privileged access. For example, BERT [40] was built on top of one Nvidia P100 GPU, Yelp dataset, and more than two days of fine-tuning. In contrast, Calico explores an innovative approach of using LLMs—conducting knowledge screening and subsequent calibration without the expensive fine-tuning process. This approach could potentially make AI-assisted research more accessible.

3 Calico

Calico aims to enhance the performance of LLMs in cross-domain code tasks. As shown in Figure 2, Calico contains three novel components that work in concert: (1) code structure screening by differential queries between LLMs and AST parsers (Section 3.1); (2) source-to-source transformation to incorporate overlooked information into original programs (Section 3.2); and (3) *what-if* analysis to diagnose performance deficiencies (Section 3.3). Its three-pronged approach builds on two key insights. First, LLM's performance can be improved by supplementing overlooked knowledge. Second, we can ensure that prompts remain concise by integrating only the most significant knowledge, which can concurrently serve as an explanation for performance deficiencies.

3.1 AI Mastery Screening

Successfully executing code tasks necessitates an in-depth understanding of various code aspects, where code structure is a critical foundation. This structure is typically represented as an AST, with nodes representing syntactic constructs such as variables and operators, and edges indicating the relationship between these constructs. ASTs have been extensively used in prior research for various purposes, including code representation [8], clone detection [26, 54], and software evolution analysis [49]. Leveraging this insight, Calico starts its process by evaluating LLM proficiency in understanding code structures. Calico automates this process in two steps.

Step 1. Query Generation. Calico uses treesitter [1] parsing library to analyze the structure of source code. Treesitter constructs an AST for the given code and supports AST-based code editing. It is compatible with over 100 programming languages, which substantially enhances Calico's ability to process extensive codebases.

In treesitter's ASTs, nodes corresponding to expressions follow particular patterns that are represented as S-expressions [3]. These patterns are related to the language parsers supported within treesitter. For instance, a C language parser contains examples [4] of input source code and the expected output AST, written as an S-expression. The binary expression `1 + 2` in a C program is represented as a `binary_expression` node, which branches into two `number_literal` child nodes. This structure conforms to the pattern `(binary_expression (number_literal) (number_literal))`. By identifying such patterns within ASTs, Calico locates the desired code structures and generates queries to extract information from these code regions. Calico analyzes 5 distinct types of code structures, including:

- **Basic Structure.** Calico identifies basic structures such as (1) preprocessor directives (*e.g.*,`#include <stdio.h>`), (2) library imports (*e.g.*,`import ObjectDoesNotExist` in Line 1 of Figure 1), (3) annotations (*e.g.*,`@login_required` in Line 3 of Figure 1), and (4) comments. These elements are essential for code understanding. For example, given a macro definition `#define PI 3.14`, LLMs should recognize that all occurrences of `PI` are consistently replaced with 3.14. Calico finds these structures by matching patterns such as `(preproc_include)`, `(import_declaration)`, `(annotation)`, and `(comment)` in ASTs.
- **API Call.** API calls are the standard method of interacting with external components (*e.g.,* libraries in `stdio.h`). Understanding API usage is beneficial for various tasks. For example, when a code uses `strcpy` without boundary checks, LLMs must recognize the potential security risks such as buffer overflows. Calico identifies API calls by searching for the `(call_expression)` pattern within ASTs.
- **Control Flow.** Control flow indicates the order of a program's statement execution, which is often determined by conditional statements (*e.g.*,`if` statements) and loops (*e.g.*,`while` loops). Understanding control flow is critical to comprehending the program's logic, locating bugs, and identifying optimization opportunities. Calico detects control flow by searching ASTs for key statements such as `(if_statement)`, `(switch_statement)`, `(for_statement)`, *etc.*

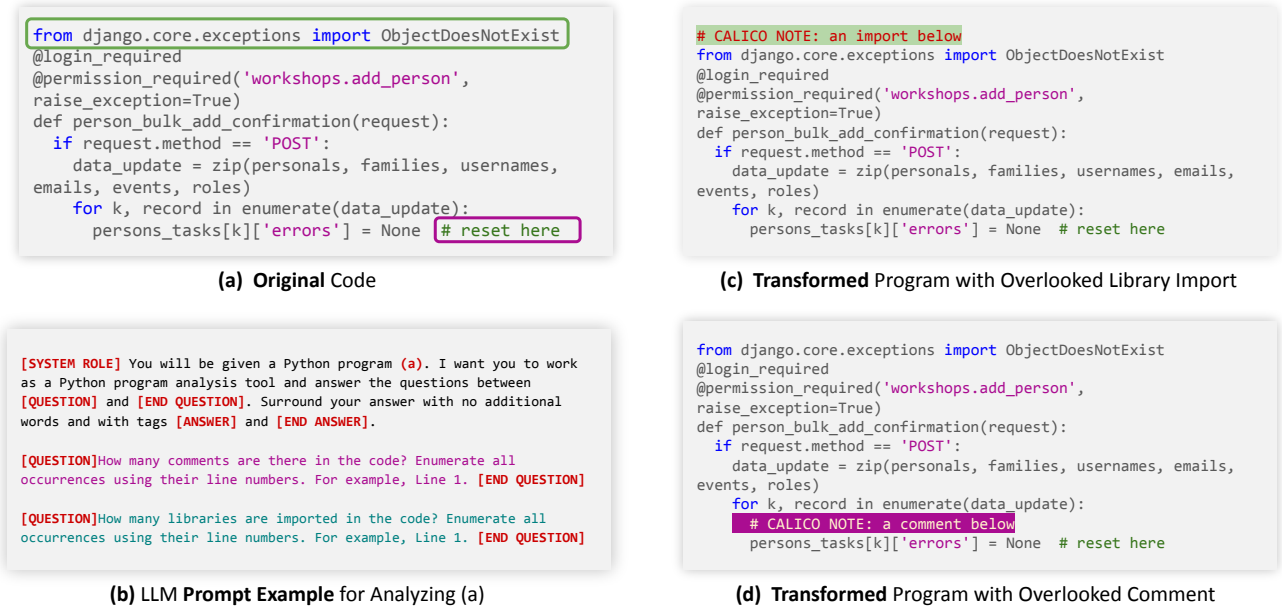


Figure 3: LLM Prompt Example and S2S Transformation.

- **Variable and Type.** Variables and data types are fundamental to programming because they define and manage data essential for program execution. LLMs need to understand them for optimizing code, finding bugs, and solving other problems. For example, LLMs should recognize type errors when dividing a number by a string in the expression `10 / "1"`. Calico finds variables and their types in ASTs, located in `(identifier)` and `primitive_type` respectively.
- **Expression.** Expressions, such as computations and assignments, involve variables (*i.e.,* operands) and operators. They are crucial in assessing conditions in programs. Calico locates expressions by matching the pattern `(expression_statement (<operator>))`, where `<operator>` represents different operators such as `binary_operator`.

Calico automatically generates queries for each of the specified types. Calico assigns a system role to the LLM and compiles a list of questions covering all AST nodes. These questions are structured according to our specific format, marked by the tags `QUESTION` and `END QUESTION`, as shown in Figure 3 (b).

Step 2. Result Collection and Comparison. After generating queries, Calico extracts their ground-truth information by analyzing AST attributes. For example, to identify comments, Calico matches the `(comment)` pattern and then extracts the content from each AST node by reading the attribute `text`. It also determines the comment's location using `start_point` and `end_point` attributes. As shown in Figure 1, Calico extracts details about an inline comment in Line 6 of `add_confirmation.py`. This comment spans from columns 48 to 60, with the content `# reset here`.

To evaluate the code comprehension of LLMs, Calico prompts LLMs with all questions and checks their responses against the correct answers. As shown in Figure 3 (b), line numbers and the tags `ANSWER` and `END ANSWER` are used to simplify output parsing. For example, Calico asks LLMs to list the line numbers of identified code structures. It then compares these line numbers against those derived from the AST attribute `start_point`. This process verifies LLM's ability to identify code elements within a given code structure. For example, in Figure 1, after promoting ChatGPT with 25 questions, it incorrectly answers 2 questions, one about library imports and another about comments. Calico identifies that ChatGPT overlooks the library import in Line 1 and the comment in Line 6, which correspond to the highlighted AST nodes in Figure 1.

Takeaway. Calico generates queries for 5 specified types of code structures to test LLMs' code understanding. Its innovative design is *general* and can be *readily extended* to a wide range of coding knowledge by incorporating new queries, such as those for dataflow analysis. We explore a novel approach to utilizing LLMs, evaluating their comprehension, and pinpointing the insights they miss.

3.2 Knowledge Calibration

Prior efforts to improve LLM's performance have included fine-tuning LLMs with task-specific data and utilizing in-context learning with provided examples. However, these two approaches provide a broad understanding of tasks such as defect detection, rather than concentrating on the actual, specific program in question. Thus, LLMs' understanding of a specific question remains unclear. In other words, even with fine-tuning or in-context learning, there is no guarantee that LLMs can completely understand the program `add_confirmation.py` in Figure 1.

Based on the insight that the information inherent in the current question is critical for completing the task, Calico incorporates details that are specific to the current question but missed by LLMs into the source code being analyzed. Such integration refocuses LLM's attention on these specific areas. To aid this process, Calico employs AST-based source-to-source transformation by leveraging

Algorithm 1: Deficiency Diagnosis

Input: $K = \{k_1, k_2, ..., k_n\}$, all overlooked knowledge; P, the original program
Output: K', the minimum set of overlooked knowledge to improve LLM performance; P', the program augmented with K'

```
1  begin
2      combinations ← ∅
3      Add an empty set into combinations
4      while combinations ≠ ∅ do
5          C ← the smallest combination in combinations
6          Pop out C from combinations
7          if test_llm(C, P) passes then
8              K' ← C
9              P' ← P + K'
10             Terminate the while loop
11         else
12             foreach k_i ∈ K do
13                 C' ← C + {k_i}
14                 Add C' into combinations
15             end
16         end
17     end
18     return K', P'
19 end
```

treesitter. It annotates each AST node that was previously overlooked with a preceding comment node. This annotation serves as a reminder that the subsequent code requires extra attention.

Figure 3 (a) presents a program that enables users to update their information such as personals, families, and usernames. After screening ChatGPT's responses to query examples in Figure 3 (b), Calico identifies that ChatGPT overlooked AST nodes corresponding to a library import (highlighted in the green box) and a comment (highlighted in the purple box). Subsequently, Calico performs code refactoring to insert comment-based annotations for knowledge calibration. Such annotations are prefixed with `"#CALICO NOTE:"` For example, Calico generates a version of the program with an annotation reminding of the library import in Figure 3 (c) and another version highlighting the comment in Figure 3 (d). These annotations direct attention to the specific areas that require further investigation.

3.3 Deficiency Diagnosis

One straightforward way of calibrating the overlooked knowledge is to add all of them into LLMs. However, this method poses two problems: (1) it does not offer insights into the model's specific deficiencies in processing the current task; and (2) it may exceed the token limitations inherent in LLMs which require concise prompts that are still informative.

There have been developments of counterfactual reasoning techniques in explainable AI [12] to pinpoint the minimal yet essential changes needed to alter a model's prediction. Calico leverages this insight through *what-if analysis* to investigate how filling different knowledge gaps affects LLM responses. In fact, Calico answers "What would happen to the LLM's response if we were to fill alternative knowledge gaps?" By doing so, we aim to *find the most significant knowledge that helps LLM complete the current task*, which concurrently explains previous deficiencies. In other words, if addressing a particular knowledge gap corrects an LLM's response, it is likely to be the underlying root cause of the previous error.

Algorithm. Algorithm 1 outlines the overall diagnosis process to isolate a minimal set of overlooked knowledge. Starting with all overlooked knowledge and the original program (Line Input), Calico employs breadth-first search and integrates the essential subset into the original program (Line Output). For this, Calico iteratively constructs powersets of the overlooked knowledge (Line 13-14) to include all possible combinations. It evaluates each combination from smallest (Line 5) to largest until a subset corrects the LLM's response (Line 7). As such, Calico successfully identifies the required minimum knowledge (Line 8) and augments the original program by integrating this knowledge into the original program (Line 9). At this point, Calico terminates the search-based diagnosis process (Line 10). If the current combination fails to correct an LLM's response, Calico explores other combinations (Line 11-16). Calico generates larger combinations by adding new overlooked knowledge to the current combination (Line 13) and stores the newly constructed combinations for future analysis (Line 14).

An execution of Algorithm 1 is shown in Figure 1. Calico starts from the smallest combination `{N1}`, and the resulting augmented code is shown in Figure 3 (d) where a comment annotation is highlighted in purple. Since ChatGPT's response remains incorrect, Calico proceeds to the combination `{N2}` and generates the augmented code as shown in Figure 3 (c) where the overlooked library import is highlighted in green. ChatGPT provides the correct answer after prompting with this newly augmented code. Calico thus concludes that the library import information is essential for this task. Then, Calico terminates the search process, and the larger combination `{N1, N2}` is pruned from the search space.

4 Evaluation Results

We seek to answer the following research questions.

RQ1 Can LLMs truly understand code structures?
RQ2 How effective is Calico in improving LLMs' performance in code-related tasks by performing knowledge calibration to fill knowledge gaps?
RQ3 What benefits can Calico obtain by performing counterfactual reasoning based diagnosis to add *an essential set of overlooked knowledge*, compared to adding all overlooked knowledge?
RQ4 How does Calico compare to fine-tuned LLMs in code tasks?

Datasets. We extensively evaluate Calico on *five* open-source, widely-used datasets, including Bugs2Fix [60], CodeSearchNet [28], Devign [77], ETHPy150Open [30], and InferredBugs [29], with a total of **8,938** programs as shown in Table 1. These datasets span *four* most commonly used programming languages—C, Java, JavaScript, and Python. We use them across *three* code tasks—bug detection, code summarization, and program repair.

- **Bug Detection.** This task leverages LLMs to identify if the given source code contains bugs or not. Thus, the output is a binary detection value such as `True` or `False`, where `True` indicates the presence of bugs and `False` means their absence. To evaluate Calico's capabilities of handling different programming languages,

Table 1: Subject Programs.

Task	ID	Dataset	Language	Description	# of Programs
Bug Detection	BD1	Devign [77]	C	Identify if a function is vulnerable	1231
	BD2	InferredBugs [29]	Java	Identify if a method is buggy	1004
	BD3	ETHPy150Open [30]	Python	Identify if a binary operator is wrongly used as another	1690
Code Summarization	CS1	CodeSearchNet [28]	Java	Given a code, generate its natural language docstring	1277
	CS2	CodeSearchNet [28]	JavaScript	Given a code, generate its natural language docstring	1031
	CS3	ETHPy150Open [30]	Python	Given a function, generate its natural language docstring	410
Program Repair	PR1	Bugs2Fix [60]	Java	Refine the code by fixing bugs	1072
	PR2	ETHPy150Open [30]	Python	Fix the bugs where some variables are misused as another	1223
Total					**8938**

we follow prior work [29, 30, 41, 50, 51] and select three widely used datasets, as reported in Table 1. They include Devign [77] for C, InferredBugs [29] for Java, and ETHPy150Open [30] for Python.

- **Code Summarization.** This task creates concise natural language descriptions for the source code [35, 43]. Thus, the output is natural language. We follow prior work [30, 41, 50, 51] and select three commonly used datasets to evaluate Calico, as reported in Table 1. They include CodeSearchNet [28]'s Java benchmarks and JavaScript benchmarks, and ETHPy150Open for Python.
- **Program Repair.** This task aims to produce patches for buggy code [76]. Thus, the output is the source code. We follow prior work [30, 41, 50, 51] and select two widely used datasets for evaluation, as reported in Table 1. They include Bugs2Fix [60] for Java and ETHPy150Open for Python.

Baselines. We use Calico to enhance the effectiveness of ChatGPT [2] (based on the default GPT-3.5 model) and compare our results with the following baselines.

- **Calico-NoDiagnosis.** This option disables the counterfactual reasoning based diagnosis from Calico. It adds all the overlooked knowledge into the source code for knowledge calibration.
- **VanillaGPT.** This option is the vanilla ChatGPT without Calico for knowledge calibration and deficiency diagnosis. We initiate a new session with ChatGPT for every assessment (*i.e.,* one session only contains one prompt and the corresponding response) to avoid potential bias from prior conversions.
- **CoTexT.** This is the SOTA fine-tuned LLM for bug detection tasks according to prior studies [50, 51]. We use its Huggingface checkpoint [5] and deploy it to our local server for evaluation.
- **CodeT5.** This is the SOTA fine-tuned LLM for code summarization and program repair tasks according to prior studies [50, 51]. We use its Huggingface checkpoint [6] and deploy it to our local server for evaluation.

Metrics. Code tasks yield various types of outputs, *i.e.,* prediction values, source code, and text, as discussed in Section 2.1. We describe our evaluation metrics as follows.

- **Accuracy.** We follow previous work [50, 51, 68] and use accuracy to evaluate the results of bug detection tasks. We obtain accuracy scores by comparing LLM's bug detection prediction (*i.e.,* buggy or not) with the ground-truth results available in the datasets.
- **BLEU.** We follow previous work [37, 50, 51, 61, 62] and use BLEU [53] (*i.e.,* bilingual evaluation understudy) to evaluate the generated docstring in code summarization tasks. BLEU assesses the quality of generated text by calculating the overlaps between the reference and generated texts. BLEU scores range from 0 to 1, where a higher value is better and 1 indicates an exact match with the reference. We obtain BLEU scores by comparing LLM's generated docstring with the ground-truth one in the datasets.
- **CodeBLEU.** We following previous work [37, 50, 51, 61] and use CodeBLEU [55] to evaluate the generated code in program repair tasks. CodeBLEU considers the similarity in ASTs and dataflows between the generated and reference code. A CodeBLEU value ranges from 0 to 1, where higher values indicate better similarity. We obtain CodeBLEU scores by comparing LLM's generated code with the ground-truth one in the datasets.

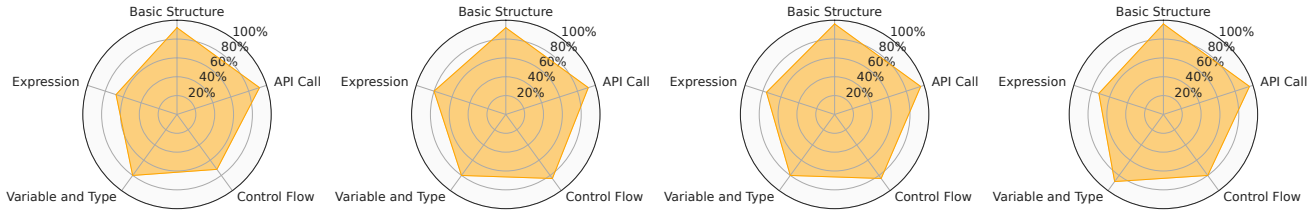
Experimental Environment. We conducted all our experiments on a Linux server running Ubuntu 20.04 LTS. It is equipped with Intel(R) Xeon(R) 2.00 GHz 96-core CPU, 354 GB RAM, and 8 Tesla V100 SXM2 16 GB GPUs. The AST analysis of source code is based on treesitter version 0.20.4. Calico requires Python version 3.8.10. SOTA comparisons need CUDA version 12.0.

4.1 RQ1: Can LLMs understand Code

To assess the proficiency of VanillaGPT in code comprehension, we prompt VanillaGPT with source code and the corresponding queries generated by Calico. We use all 8,938 programs listed in Table 1 and include one single program in each prompt. For each program, we use Calico to generate queries that cover all of the five types of queries discussed in Section 3.1. We measure the average percentage of correct responses to these queries of each type for each programming language.

Figure 4 shows VanillaGPT's code comprehension on our selected datasets, which include C, Java, JavaScript, and Python programs. Each radar chart illustrates VanillaGPT's comprehension performance, representing the percentage of correct responses for a language. These charts consist of five radials, with each radial representing a specific code structure that corresponds to one query type. The position of the node on each radial reflects the correct rate (ranging from 0% to 100%) for the corresponding structure. The larger the area of the pentagon, the more accurate the VanillaGPT.

In general, VanillaGPT demonstrates limited comprehension of code structures, as it does not achieve 100% correctness in any structure. VanillaGPT exhibits a good understanding of basic structures such as comments and annotations, as well as API calls like `isinstance()` and `len()`. Across all languages, VanillaGPT



(a) C Programs. (b) Java Programs. (c) JavaScript Programs. (d) Python Programs.

Figure 4: ChatGPT's Code Comprehension for C, Java, JavaScript, and Python Programs.

achieves a correct rate exceeding 90% for these two structures. In cases where unidentified structures arise, we re-prompt VanillaGPT with the same query, resulting in successful identification. For example, for the program `Achilles` in the dataset InferredBugs, when we prompt VanillaGPT "How many APIs are called in the code?" for the first time, VanillaGPT missed `getPropertyName()` but correctly identified it upon repeating the same query. This indicates VanillaGPT's capabilities of comprehending basic structures and API calls, albeit with occasional oversight that is probably due to the need for further iterations to refine recognition.

For the remaining three code structures, including control flow for recognizing statements like `while` loops, variables and types for identifying variables, and expressions for operand and operator identification, VanillaGPT's correct rates are around 80%. VanillaGPT performs less effectively in identifying expressions, with correct rates ranging from 68% to 80% across the four languages. For example, in the ETHPy150Open dataset, VanillaGPT initially ignored the nested expression `(35*x**4/8 - 15*x**2/4 + S(3)/8)/(2*(y + 1))` and did not classify it even after re-prompting with the query "How many expressions are there in the code?" Only upon explicitly asking VanillaGPT "Is this an expression? `(35*x**4/8 - 15*x**2/4 + S(3)/8)/(2*(y + 1))`" did it finally recognize it as such. This underscores the need for repeated exposure to identify these more intricate structures. In contrast to basic structures identifiable through keywords like `import` or tokens such as #, or API calls recognizable by parentheses, control flow, variables, and expressions may necessitate additional information such as condition evaluation and operation order comparison. These complexities can pose challenges in understanding these structures.

Among the four languages examined, VanillaGPT exhibits similar comprehension across Java, JavaScript, and Python, with average correct rates for 5 structures at 85.6%, 88%, and 86.4%, respectively. However, the average correct rate for understanding C structures is lower, at 80.8%. This is potentially because of the comparatively smaller number of C programs in VanillaGPT's training dataset, limiting its familiarity with C programming structures.

Summary 1

VanillaGPT understands 90% of basic structures and API calls, while around 80% of control flow, variables, and expressions. It understands Java, JavaScript, and Python better than C.

4.2 RQ2: Effectiveness of Knowledge Calibration

To assess the effectiveness of Calico's knowledge calibration to fill knowledge gaps, we compare the performance of VanillaGPT and Calico on three code tasks, *i.e.,* bug detection, code summarization, and program repair, using the datasets listed in Table 1. We complete the experiments in two rounds: (1) round 1 involves evaluating VanillaGPT by prompting it with the original source code and asking it to complete the corresponding task; (2) round 2 involves evaluating Calico: first, we use Calico to screen VanillaGPT for identifying any overlooked knowledge within each program; then, we use Calico to add the minimum set of overlooked knowledge into the original program; finally, we prompt VanillaGPT with this augmented program to complete the corresponding task. We calculate average accuracy for bug detection, BLEU for code summarization, and CodeBLEU for program repair.

We show the benefits of knowledge calibration in Figure 5. Across all three tasks, Calico consistently outperforms VanillaGPT. On average, Calico enhances accuracy for bug detection by 10%, improves BLEU scores for code summarization by 15.35%, and boosts CodeBLEU scores for program repair by 19.93%. These results indicate that knowledge calibration of Calico enhances VanillaGPT's understanding of code structures.

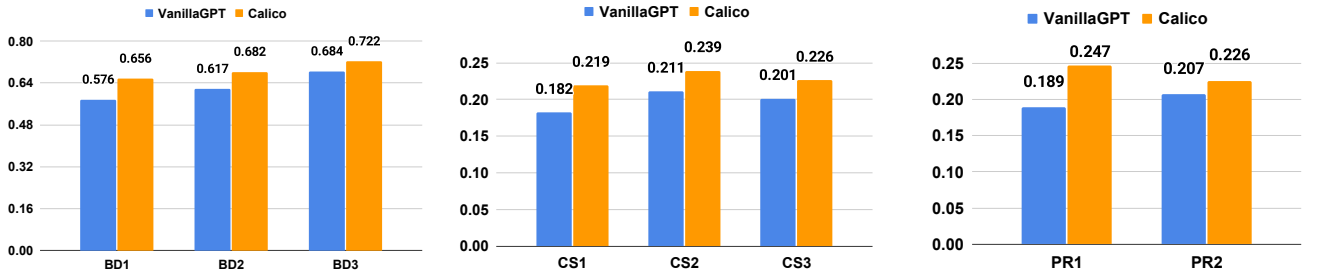
For example, a Python program in ETHPy150Open has a bug where the incorrect binary operator `and` is used instead of the correct `or`. Initially, VanillaGPT failed to identify this bug. After the screening, Calico discovered that VanillaGPT had overlooked the `if add_missing and not item_present` statement. Calico annotated this `if` statement to draw extra attention from VanillaGPT. Subsequently, when prompting again with the augmented program, VanillaGPT correctly recognized this program as buggy.

Summary 2

VanillaGPT's code understanding can be improved by integrating overlooked knowledge. On average, it achieves 10% improvement for bug detection, 15.35% for code summarization, and 19.93% for program repair.

4.3 RQ3: Benefits of Deficiency Diagnosis

To assess the benefits of adding a *minimum* set of overlooked knowledge instead of integrating *all* the overlooked knowledge, we compare Calico against a downgraded version Calico-NoDiagnosis.



(a) Acccuracy Benefits for Bug Detection. **(b) BLEU Benefits for Code Summarization.** **(c) CodeBLEU Benefits for Program Repair.**

Figure 5: Benefits of Knowledge Calibration.

Calico-NoDiagnosis augments code by adding all the overlooked knowledge. Since Calico checks all knowledge combinations from smallest to largest in an iterative manner, we measure the average program size in terms of lines of code (LOC) when Calico achieves the same performance as Calico-NoDiagnosis.

We report the program size in Table 2. Counterfactual reasoning based deficiency diagnosis in Calico leads to an average reduction rate of 8.39%. Calico achieves a higher reduction rate for smaller programs such as the Python programs in PR2 (ETHPy150Open) that contain a single function. Based on our experiments, Calico can achieve up to 14.81% reduction rate with deficiency diagnosis. This is because adding annotations can substantially increase their code size relative to their original small size. In contrast, for larger programs like the C programs in BD1 (Devign) and Java programs in BD2 (InferredBugs), these annotations have a lesser impact on program sizes compared to their original sizes.

Take the Java program `DragonProxy` of 186 LOC from the dataset InferredBugs as an example. Its method `initialize()` has a bug of resource leak where the resource `fos` is not released after acquiring from `FileOutputStream()`. VanillaGPT overlooked the API call `FileOutputStream()`, resulting in its inaccurate bug detection. Upon augmentation by Calico-NoDiagnosis, all overlooked knowledge was integrated into `DragonProxy`, resulting in a new program of 197 LOC. However, Calico incorporated only essential knowledge, including the overlooked API call, generating an augmented code of 192 LOC. In this way, Calico preserves sufficient information to explain VanillaGPT's deficiencies while maintaining the LLM prompt concise.

Summary 3

Calico explains the deficiencies by identifying a minimum set of overlooked knowledge and generates shorter augmented code compared to Calico-NoDiagnosis. On average, Calico achieves a reduction rate of 8.39%.

4.4 RQ4: Effectiveness Compared with Fine Tuning

With information encoded into parameters, fine-tuning LLMs is still one of the most effective approaches to teaching LLMs task-specific knowledge. To assess the effectiveness of Calico compared with fine-tuned LLMs, we compare Calico against SOTA LLMs according to the prior study [51], *i.e.*,CoTexT for bug detection and CodeT5 for code summarization and program repair.

Table 3 reports our comparison results. In general, SOTA LLMs outperform Calico. However, performance differences are small, ranging from 0.009 to 0.102. Considering the substantial amount of new data and computing resources required for model fine-tuning, Calico achieves notable performance with knowledge calibration. In particular, Calico outperforms the SOTA CodeT5 on PR1, enhancing the CodeBLEU score by 0.006. Calico achieves a large improvement, from 0.527 to 0.656, on BD1. This is probably because CoTexT is not

Table 2: Benefits of Deficiency Diagnosis. Calico-ND stands for Calico-NoDiagnosis.

Dataset	Average Size (LOC)			Reduction Rate
	Original	Calico-ND	Calico	
BD1	239	267	259	3.00%
BD2	366	436	417	4.56%
BD3	19	26	23	13.04%
CS1	87	102	96	6.25%
CS2	25	31	27	**14.81%**
CS3	48	57	53	7.55%
PR1	17	23	22	4.54%
PR2	13	17	15	13.33%

Table 3: Calico vs. Fine-tuned LLMs.

Metric	Dataset	SOTA		Calico
		CoTexT	CodeT5	
Accuracy	**BD1**	0.527	N/A	**0.656**
	BD2	**0.712**	N/A	0.682
	BD3	**0.749**	N/A	0.722
BLEU	**CS1**	N/A	**0.313**	0.219
	CS2	N/A	**0.341**	0.239
	CS3	N/A	**0.239**	0.226
CodeBLEU	**PR1**	N/A	0.241	**0.247**
	PR2	N/A	**0.235**	0.226

Table 4: Calico Average Latency.

Task	Screening	Calibration	Total
Bug Detection	3.7s	26.6s	**30.3s**
Code Summarization	1.8s	9.7s	**11.5s**
Program Repair	0.6s	4.9s	**5.5s**

trained with C datasets such as BD1 and may lack familiarity with the C language to effectively complete the bug detection task.

Compared to fine-tuned models, Calico introduces extra overhead in its AI mastery screening and knowledge calibration. We report the average overhead of Calico in Table 4. For our evaluation datasets, on average, Calico needs 2 seconds for identifying the overlooked knowledge, which is a one-time effort, and 13 seconds for generating the augmented code with a minimum set of overlooked knowledge. However, Calico does not need model fine-tuning. Although the exact time required for fine-tuning CoTexT and CodeT5 is not explicitly reported in the prior study [51], we can estimate from prior work [59] that fine-tuning an LLM with 220 million parameters such as CoTexT and CodeT5 can take a whole day on NVIDIA A100 GPU. This overhead significantly exceeds that of Calico, while Calico demonstrates similar performance to SOTA LLMs. In conclusion, the screening and knowledge calibration in Calico introduce minimal overhead.

Summary 4

Calico achieves comparable results to SOTA LLMs with **no** fine-tuning used. The performance differences are less than 0.102, and Calico outperforms CodeT5 on PR1. Screening and knowledge calibration take less than 30 seconds, *avoiding days of model fine-tuning.*

5 Related Work

LLMs in SE. Recent work [16–18, 25, 36, 57, 71, 78] have explored the application of LLMs to solve SE problems. Deng et al. [15] use two LLMs, Codex and InCoder, to generate DL programs for testing DL libraries PyTorch and TensorFlow. Yang et al. [73] locate buggy lines of code by fine-tuning the LLM CodeGen. Another line of work [51, 58] leverages SE approaches to better understand the capabilities of LLMs. Li et al. [38] investigate the feasibility of extracting LLMs' code abilities such as code synthesis and code translation by launching imitation attacks. Wu et al. [70] analyze the limits of ChatGPT in software security applications such as decompilation and patching. Calico, however, investigates a new way of using vanilla LLMs by first examining how well LLMs understand source code and then filling knowledge gaps to improve LLM performance.

Prompt Engineering. To address the limitations of model fine-tuning, many prompt engineering approaches have emerged [10, 44, 67, 69]. When leveraging prompt engineering, users can incorporate task descriptions and/or demonstration examples into the prompt to equip LLMs with task-specific knowledge without any parameter updates. For example, to test DL libraries, Deng et al. [16] employ in-context zero-shot learning to autocomplete a partial bug-triggering program and in-context few-shot learning to generate a new code snippet based on the example programs. To recover software architecture, Rukmono et al. [56] use GPT-4 to emulate deductive reasoning via chain-of-thoughts prompting. These approaches aim to provide additional task context and fill the knowledge gaps by integrating more information into prompts. Our work Calico shares a similar motivation. However, instead of offering general context, Calico provides information directly relevant to the source code under analysis. It begins by identifying essential knowledge gaps within LLM's current comprehension and then integrates specific information necessary to bridge these gaps effectively.

Automated Code Refactoring. Since pioneering work on automated refactoring in early 1990s [20, 45, 52], recent research finds that real-world refactorings often fail to preserve semantics [32, 33], reply on manual intervention [63], prone to errors [31, 48], and exceed the capabilities of existing refactoring engines. A study with professional developers finds that nearly 12% of refactorings are motivated to improve performance [33]. Calico builds on this foundation but repurposes it to knowledge calibration in the new landscape of LLM for SE. By systematically and automatically adding AST nodes of comments, Calico ensures semantic preservation while highlighting information overlooked by LLMs.

Counterfactual Reasoning. The black-box nature of deep neural networks hinders their explainability and thus challenges their applications in areas such as healthcare and military [13]. To address this problem, explainable AI [9, 13, 65] aims to promote techniques that generate explanations of AI predictions. Recent work [12] uses counterfactual reasoning to figure out what changes are necessary for obtaining specific model predictions, *i.e.,* to answer a what-if question that what would have been the effect of model predictions if we had taken action or vice versa. Calico extends this approach and adapts it to reveal why LLMs are struggling with code tasks. By systematically adding different combinations of overlooked knowledge, Calico identifies the minimum necessary knowledge to improve LLM performance, which serves as an explanation for their deficiencies.

Delta Debugging. Delta debugging [74, 75] stands as the fundamental work of automated test or code reduction. It aims to locate the minimum failure-inducing changes between two program versions. Recent work [22, 23, 34, 47] has sought to enhance the efficiency and effectiveness of its core algorithm, *ddmin*. ProbDD [66] integrates testing history to assign probabilities to code elements, prioritizing those with a higher likelihood of being retained in the reduced program. All of this existing work executes different variants to find the minimum parts that *retain* the symptom (*e.g.,* program failures). In contrast, Calico mutates programs to find the minimum changes that *alter* the symptom (*e.g.,* wrong LLM responses), indicating that such changes are critical to the symptom and can be seen as the root cause of LLM deficiencies.

6 Threats to Validity

Models. We use ChatGPT to evaluate Calico. Since its training data remains disclosed, the selection of evaluation datasets may impact the quantitative results. To mitigate this potential bias, we have

selected five diverse datasets spanning four programming languages and applied three evaluation metrics. Expanding the evaluation to include more LLMs could facilitate further assessment.

Prompts. The performance of an LLM can be sensitive to prompts and tasks. We follow prior work [21, 39] to design our prompts of queries, explicitly specifying the system role, task description, programming language, source code, and code structures for analysis to mitigate bias and provide clear information and instructions. Model randomness could be addressed via repeated prompting.

Queries and AST Parsers. The knowledge overlooked by LLMs is identified based on their responses to Calico's generated queries. We have designed queries for five kinds of code structures. The quantitative results of code comprehension may change as additional code structures or other types of analysis techniques are included.

7 Discussion

Calico evaluates AST-level knowledge as a first step. The insight is that code structure is fundamental to completing code tasks. Since this structure is inherently present in the code itself, Calico operates under the assumption that AST parsers can guide its process in practical, real-world settings. Deficiency diagnosis is not intended to improve LLM performance directly; instead, it seeks to explain previous deficiencies. Calico employs a what-if analysis to investigate the impact of filling different knowledge gaps. For example, when repairing buggy code, an LLM might initially propose a fix that results in ten compilation errors. After Calico supplements some previously overlooked knowledge, the LLM's subsequent fix may reduce the errors to nine. This improvement indicates that the incorporated knowledge effectively addresses some of the issues, thereby explaining the resolved errors.

Calico is independent of knowledge scopes. Its current functionality is limited by the capabilities of AST parsers in processing syntactic knowledge. However, it can easily extend to other knowledge domains. For instance, Calico can integrate pointer analysis tools for tasks to identify and address unauthorized memory access issues. In scenarios requiring code refactoring due to updates in external libraries, Calico can leverage tools that track library evolution, such as API changes, and incorporate this information into the code accordingly.

8 Conclusion

LLMs have gained substantial traction in code tasks such as bug detection, code summarization, and program repair. However, they still encounter challenges in effectively managing these tasks. To address this problem, we propose Calico that identifies the knowledge gaps in LLM's program comprehension and integrates a minimum set of overlooked knowledge necessary for LLM performance enhancement into the original program. Leveraging knowledge calibration, Calico achieves improvements of up to 20% for vanilla ChatGPT in bug detection, code summarization, and program repair tasks, demonstrating comparable proficiency compared to SOTA fine-tuned LLMs without the need for model fine-tuning. Deficiency diagnosis contributes to an average program reduction of 8.39%, compared to integrating all knowledge into the source code.

Acknowledgement

The participants of this research are in part supported by Cisco grant, Regents Faculty Fellowship offered by UCR Academic Senate, and Omnibus Research Award.

References

[1] 2024. https://tree-sitter.github.io/tree-sitter/.
[2] 2024. https://chat.openai.com/.
[3] 2024. https://tree-sitter.github.io/tree-sitter/using-parsers#pattern-matching-with-queries.
[4] 2024. https://github.com/tree-sitter/tree-sitter-c/blob/master/test/corpus/statements.txt.
[5] 2024. https://huggingface.co/razent/cotext-2-cc.
[6] 2024. https://huggingface.co/Salesforce/codet5-base.
[7] 2024. GitHub Copilot. https://github.com/features/copilot.
[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (jan 2019), 29 pages. https://doi.org/10.1145/3290353
[9] Plamen P Angelov, Eduardo A Soares, Richard Jiang, Nicholas I Arnold, and Peter M Atkinson. 2021. Explainable artificial intelligence: an analytical review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 11, 5 (2021), e1424.
[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165
[11] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 18–30. https://doi.org/10.1145/3540250.3549162
[12] Yu-Liang Chou, Catarina Moreira, Peter Bruza, Chun Ouyang, and Joaquim Jorge. 2022. Counterfactuals and causability in explainable artificial intelligence: Theory, algorithms, and applications. *Information Fusion* 81 (2022), 59–83. https://doi.org/10.1016/j.inffus.2021.11.003
[13] Arun Das and Paul Rad. 2020. Opportunities and Challenges in Explainable Artificial Intelligence (XAI): A Survey. arXiv:2006.11371 [cs.CV]
[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067
[15] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067
[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 70, 13 pages. https://doi.org/10.1145/3597503.3623343
[17] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128
[18] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin. 2024. Machine Learning is All You Need: A Simple Token-based Approach for Effective Code Clone Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1006–1006. https://doi.ieeecomputersociety.org/
[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020.

CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]
[20] William G. Griswold. 1991. *Program Restructuring as an Aid to Software Maintenance.* Ph. D. Dissertation. University of Washington.
[21] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. arXiv:2401.03065 [cs.SE]
[22] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation.* 31–37.
[23] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse hierarchical delta debugging. In *2017 IEEE international conference on software maintenance and evolution (ICSME).* IEEE, 194–203.
[24] Jeremy Howard and Sebastian Ruder. 2018. Fine-tuned Language Models for Text Classification. *CoRR* abs/1801.06146 (2018). arXiv:1801.06146 http://arxiv.org/abs/1801.06146
[25] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting Greybox Fuzzing with Generative AI. arXiv:2306.06782 [cs.CR]
[26] Tiancheng Hu, Zijing Xu, Yilin Fang, Yueming Wu, Bin Yuan, Deqing Zou, and Hai Jin. 2023. Fine-Grained Code Clone Detection with Block-Based Splitting of Abstract Syntax Tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023).* Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/3597926.3598040
[27] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1162–1174. https://doi.org/10.1109/ASE56229.2023.00181
[28] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG]
[29] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023).* Association for Computing Machinery, New York, NY, USA, 1646–1656. https://doi.org/10.1145/3611643.3613892
[30] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20).* JMLR.org, Article 474, 12 pages.
[31] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of Refactorings during Software Evolution. In *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering.*
[32] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12).* ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655
[33] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 1–1. https://doi.org/10.1109/TSE.2014.2318734
[34] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.* 16–22.
[35] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19).* IEEE Press, 795–806. https://doi.org/10.1109/ICSE.2019.00087
[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* 919–931. https://doi.org/10.1109/ICSE48619.2023.00085
[37] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, Singapore, Singapore,) *(ESEC/FSE 2022).* Association for Computing Machinery, New York, NY, USA, 1035–1047. https://doi.org/10.1145/3540250.3549081
[38] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Chaowei Liu, Shuai Wang, Daoyuan Wu, Cuiyun Gao, and Yang Liu. 2023. On Extracting Specialized Code Abilities from Large Language Models: A Feasibility Study. arXiv:2303.03012 [cs.SE]
[39] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815
[40] Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. 2021. AutoFreeze: Automatically Freezing Model Blocks to Accelerate Fine-tuning. arXiv:2102.01386 [cs.LG]
[41] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]
[42] Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, and Gabriele Bavota. 2023. Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization. arXiv:2312.15475 [cs.SE]
[43] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119. https://doi.org/10.1109/TSE.2015.2465386
[44] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS).*
[45] Tom Mens and Tom Tourwe. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817
[46] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent Advances in Natural Language Processing via Large Pre-trained Language Models: A Survey. *ACM Comput. Surv.* 56, 2, Article 30 (sep 2023), 40 pages. https://doi.org/10.1145/3605943
[47] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering.* 142–151.
[48] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society, Washington, DC, USA, 287–297. https://doi.org/10.1109/ICSE.2009.5070529
[49] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes* 30, 4 (may 2005), 1–5. https://doi.org/10.1145/1082983.1083143
[50] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. arXiv:2205.11739 [cs.SE]
[51] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23).* IEEE Press, 2136–2148. https://doi.org/10.1109/ICSE48619.2023.00180
[52] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph. D. Dissertation. University of Illinois, Urbana-Champaign, IL, USA. citeseer.ist.psu.edu/opdyke92refactoring.html
[53] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02).* Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135
[54] Daniel Perez and Shigeru Chiba. 2019. Cross-Language Clone Detection by Learning Over Abstract Syntax Trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* 518–528. https://doi.org/10.1109/MSR.2019.00078
[55] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]
[56] Satrio Rukmono, Lina Ochoa Venegas, and MRV Chaudron. 2024. Deductive Software Architecture Recovery via Chain-of-thought Prompting. In *International Conference on Software Engineering: New Ideas and Emerging Results.*
[57] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. arXiv:2305.00418 [cs.SE]
[58] D. Sobania, M. Briesch, C. Hanna, and J. Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR).* IEEE Computer Society, Los Alamitos, CA, USA, 23–30. https://doi.org/10.1109/APR59189.2023.00012
[59] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia

Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[60] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (sep 2019), 29 pages. https://doi.org/10.1145/3340544

[61] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2291–2302. https://doi.org/10.1145/3510003.3510621

[62] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 163–174. https://doi.org/10.1109/ICSE43902.2021.00027

[63] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on.* 233–243. https://doi.org/10.1109/ICSE.2012.6227190

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[65] Giulia Vilone and Luca Longo. 2020. Explainable Artificial Intelligence: a Systematic Review. arXiv:2006.00093 [cs.AI]

[66] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 881–892.

[67] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL]

[68] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL]

[69] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[70] Fangzhou Wu, Qingzhao Zhang, Ati Priya Bajaj, Tiffany Bao, Ning Zhang, Ruoyu "Fish" Wang, and Chaowei Xiao. 2023. Exploring the Limits of ChatGPT in Software Security Applications. arXiv:2312.05275 [cs.CR]

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[72] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3597503.3623326

[73] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3597503.3623342

[74] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA) *(SIGSOFT '02/FSE-10)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[75] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[76] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. arXiv:2310.08879 [cs.SE]

[77] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks.* Curran Associates Inc., Red Hook, NY, USA.

[78] Deqing Zou, Siyue Feng, Yueming Wu, Wenqi Suo, and Hai Jin. 2023. Tritor: Detecting Semantic Code Clones by Building Social Network-Based Triads Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 771–783. https://doi.org/10.1145/3611643.3616354

Received 2024-04-12; accepted 2024-07-03