Leveraging Large Language Model for Automatic Patch Correctness Assessment

Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo, *Fellow, IEEE*

***Abstract*—Automated Program Repair (APR) techniques have shown more and more promising results in fixing real-world bugs. Despite the effectiveness, APR techniques still face an overfitting problem: a generated patch can be incorrect although it passes all tests. It is time-consuming to manually evaluate the correctness of generated patches that can pass all available test cases. To address this problem, many approaches have been proposed to automatically assess the correctness of patches generated by APR techniques. These approaches are mainly evaluated within the cross-validation setting. However, for patches generated by a new or unseen APR tool, users are implicitly required to manually label a significant portion of these patches (e.g., 90% in 10-fold cross-validation) in the cross-validation setting before inferring the remaining patches (e.g., 10% in 10-fold cross-validation). To mitigate the issue, in this study, we propose LLM4PatchCorrect, the patch correctness assessment by adopting a large language model for code. Specifically, for patches generated by a new or unseen APR tool, LLM4PatchCorrect does not need labeled patches of this new or unseen APR tool for training but directly queries the large language model for code to get predictions on the correctness labels without training. In this way, LLM4PatchCorrect can reduce the manual labeling effort when building a model to automatically assess the correctness of generated patches of new APR tools. To provide knowledge regarding the automatic patch correctness assessment (APCA) task to the large language model for code, LLM4PatchCorrect leverages bug descriptions, execution traces, failing test cases, test coverage, and labeled patches generated by existing APR tools, before deciding the correctness of the unlabeled patches of a new or unseen APR tool. Additionally, LLM4PatchCorrect prioritizes labeled patches from existing APR tools that exhibit semantic similarity to those generated by new APR tools, enhancing the accuracy achieved by LLM4PatchCorrect for patches from new APR tools. Our experimental results showed that LLM4PatchCorrect can achieve an accuracy of 84.4% and an F1-score of 86.5% on average although no labeled patch of the new or unseen APR tool is available. In addition, our proposed technique significantly outperformed the prior state-of-the-art.**

***Index Terms*—Automatic patch correctness assessment, large language models of code, in-context learning.**

I. Introduction

AUTOMATED Program Repair (APR) has gained increasing attention and diverse APR tools have been proposed [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. Despite the significant improvements achieved in APR, existing APR tools still face a long-standing challenge: the overfitting problem [14], [15], [16], [17], [18]. Due to the absence of strong program specifications, APR tools often use test cases to validate whether a generated patch is correct or not. However, passing all the existing test cases does not ensure that the patch is indeed correct and there is no guarantee that the patch can actually repair the program. A generated patch is considered overfitting if it passes all the available test cases while it is still incorrect with respect to the intended program specification.

Identifying overfitting patches is crucial for the APR tool adoption in practice. Suppose Bob is a practitioner who is keen to use advanced APR tools. There exist multiple approaches he can employ, and each produces many patches. However, recent studies [19], [20] demonstrate that APR tools could generate more overfitting patches than correct ones, showing a high false positive rate. In addition, researchers have revealed that high false positive rates may deliver dissatisfaction and distrust to developers on automatic SE tools such as static analysis [21] and fault localization [22]. This indicates that APR tools can disappoint Bob by wasting his time with overfitting patches. Thus, it is important to detect and reduce the overfitting patches, especially for the generate-and-validate APR approaches in practice [23].

To address this issue, many approaches [4], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35] have been proposed to conduct automatic patch correctness assessment (APCA). Lin et al. [33] categorized the existing APCA approaches into two categories: (1) dynamic approaches

Manuscript received 23 March 2024; revised 16 August 2024; accepted 19 August 2024. Date of publication 30 August 2024; date of current version 14 November 2024. This work was supported by the National Research Foundation, under its Investigatorship Grant NRF-NRFI08-2022-0002. Recommended for acceptance by M. Monperrus. *(Corresponding author: Bowen Xu.)*

Xin Zhou, Kisub Kim, Hung Huu Nguyen, Junda He, and David Lo are with the School of Computing and Information Systems, Singapore Management University, Singapore 188065 (e-mail: xinzhou.2020@smu.edu.sg; falconlk00@gmail.com; huuhungn@smu.edu.sg; jundahe@smu.edu.sg; davidlo@smu.edu.sg).

Bowen Xu is with the Department of Computer Science College of Engineering, North Carolina State University, Raleigh, NC 27606 USA (e-mail: bxu22@ncsu.edu).

DongGyun Han is with the Department of Computer Science, Royal Holloway, University of London, TW20 0EX Egham, U.K. (e-mail: donggyun.han@rhul.ac.uk).

Thanh Le-Cong and Bach Le are with the School of Computing and Information Systems, The University of Melbourne, Melbourne, VIC 3010, Australia (e-mail: congthanh.le@student.unimelb.edu.au; bach.le@unimelb.edu.au).

Digital Object Identifier 10.1109/TSE.2024.3452252

which are based on running/executing the tests and (2) static approaches which are built on top of source code patterns or features. In general, dynamic approaches perform correctness assessment by either augmenting test cases using automated test generation tools such as Randoop [36] or collecting the runtime information for analysis. On the other hand, static approaches extract code patterns or features to decide the correctness. Despite the promising results, both of them still have drawbacks. The dynamic approaches are very time-consuming [28], [30] while the static approaches are more efficient but could be less precise [29]. Additionally, building certain static systems could be hard if they require crafting specialized code patterns or features. In this study, we aim to advance static APCA approaches.

Many static APCA approaches have been proposed in recent years. For example, Ye et al. [27] introduced ODS, which identifies overfitting patches by utilizing statically extracted code features at the Abstract Syntax Tree (AST) level from both the buggy code and patches generated by APR tools. Tian et al. [23] leveraged advanced code representation learning techniques such as BERT [37], to extract source code embeddings for assessing patch correctness. Recently, Tian et al. introduced Quatrain [34], which transforms the APCA task into a question-answering problem. Quatrain first learned the relationship between bug reports and patch descriptions. Subsequently, it constructed a question-answer-based classifier to assess patch correctness. Moreover, Lin et al. [33] proposed Cache, a patch correctness assessment technique that learns a context-aware code change embedding, considering program structures. Cache achieved the state-of-the-art performance and outperformed many dynamic approaches [25], [26], [29], [30], [36].

Static APCA approaches, such as ODS [27], Tian et al.'s approach [23], Quatrain [34], and Cache [33], directly extract features from patches and learn correct patterns from the labeled dataset. In prior works, static APCA approaches were primarily evaluated using k-fold cross-validation, where patches generated by different APR tools are mixed and separated into (k-1):1 for training and testing. However, the cross-validation setting has a significant limitation: for patches generated by a new or unseen APR tool, users are implicitly required to manually label a significant portion of these patches (e.g., 90% in 10-fold cross-validation) before inferring the remaining patches (e.g., 10% in 10-fold cross-validation). Suppose Bob is a practitioner eager to leverage an advanced new APR tool to fix a bug and also aims to utilize APCA approaches to filter out the overfitting patches generated by the new tool. However, the 10-fold cross-validation process implicitly necessitates that Bob manually labels 90% of the patches generated by the new APR tool. This significant manual effort may deter Bob from adopting APCA approaches.

Given the rapid emergence of new APR tools, our goal is to alleviate the burden on users by avoiding the need for manual labeling of the patches generated by these new tools before APCA approaches can predict their correctness. Additionally, many patches generated by existing APR tools have already been manually checked for correctness [23], [29]. Thus, we are motivated to ask a key question:

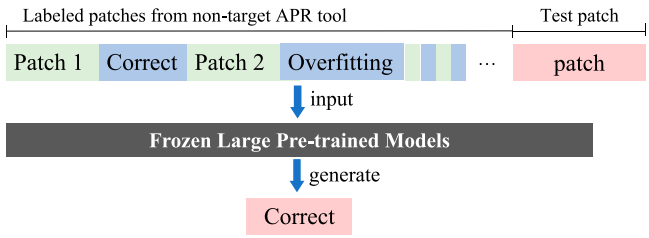


Fig. 1. An abstracted example of how to use the large pre-trained models. An unlabeled test patch of a new APR tool is concatenated by several labeled patches of existing APR tools to form the input to the pre-trained model.

*Is it feasible to utilize **labeled patches of existing APR tools** to predict the correctness of the patches generated by a **new/unseen** APR tool?*

We first explore whether the state-of-the-art APCA approaches can predict the correctness of patches generated by an unseen APR tool when labeled patches of other APR tools are available in the training data. During our preliminary experiments, we observed that state-of-the-art APCA approaches such as Quatrain [34] and Cache [33] did not yield satisfactory results. This was possibly due to a lack of labeled patches from the new or unseen APR tool for training the model. Addressing this challenge is critical for further advancing the APCA task. We report the effectiveness of these state-of-the-art approaches in Section V-A. While Ye et al. [27] have explored the setting mentioned above, they did not investigate the effectiveness of recent large language models (LLMs). This is particularly noteworthy since LLMs are known to excel in few-shot or zero-shot tasks [38] that closely align with our focus. Studying LLMs in the context of patch correctness assessment could offer valuable insights into the capabilities of these models. Additionally, in this study, we introduce a novel large language model-based solution to effectively tackle the challenge, which significantly outperforms the ODS approach proposed by Ye et al. [27].

To tackle the challenge, we propose **LLM4PatchCorrect**, which aims to enhance the effectiveness of predicting the correctness of patches generated by new or unseen APR tools. LLM4PatchCorrect employs an open-source large language model (LLM) for code, called Starcoder-7B [39], to evaluate patch correctness, without requiring fine-tuning. Technically, we directly leverage the pre-training objective of the LLM: generating the next token based on all previous tokens, to accomplish the APCA task. As shown in Fig. 1, we first prepare the model inputs by concatenating an unlabeled patch from a new APR tool with several labeled patches from existing APR tools, along with other bug-related information. We then query the LLMs to generate the next token to show its tendencies in terms of patch correctness (correct or overfitting). This allows us to apply LLMs without the need for fine-tuning since we formulate the APCA task (i.e. predicting whether a patch is correct or not) in the same format as the pre-training task (i.e., generating the next token). The utilization of LLMs in this manner is referred to as *In-context learning*.

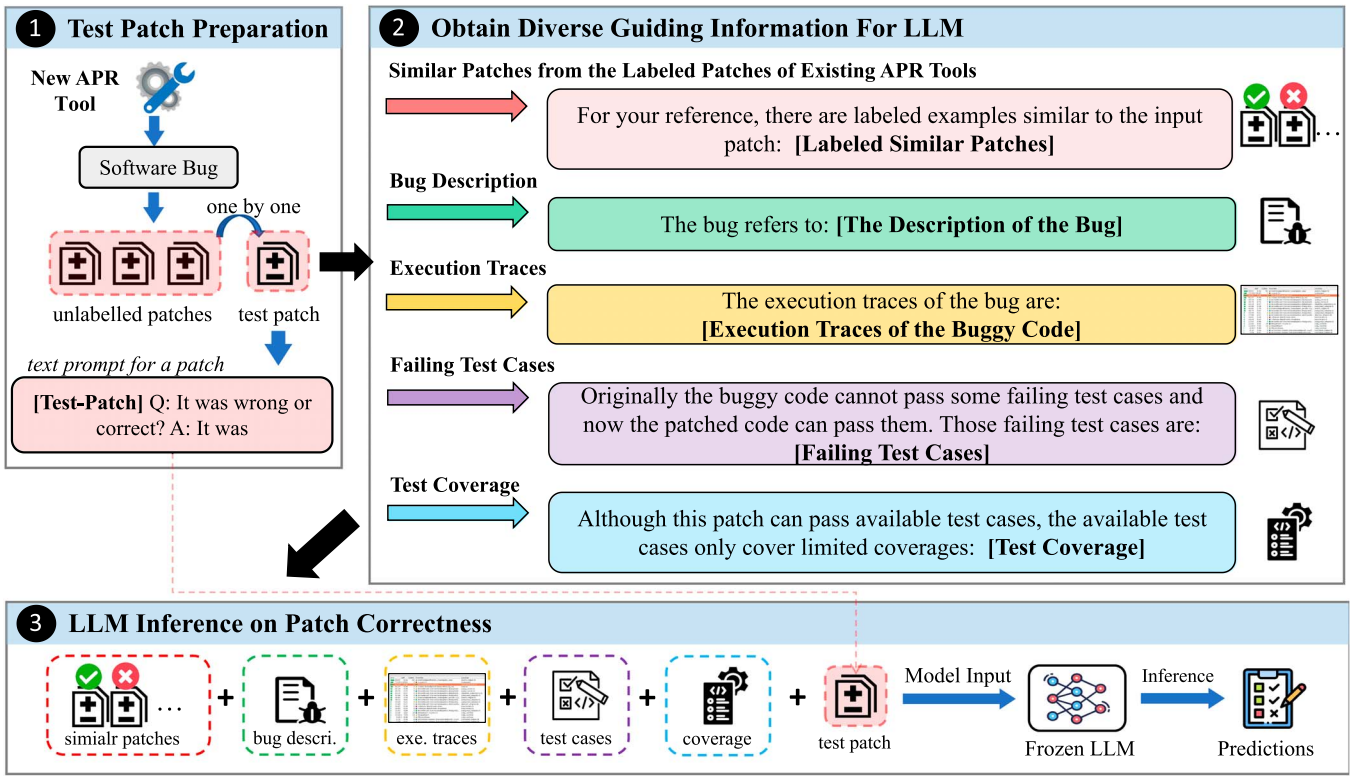


Fig. 2. Overall framework of LLM4PatchCorrect.

LLM4PatchCorrect does not encompass all labeled patches produced by existing APR tools; rather, it selects semantically similar patches. This strategy aids the large language model in providing more precise predictions for patches generated by a new APR tool. In addition to the labeled patches of existing APR tools, LLM4PatchCorrect incorporates a broader range of guiding information, as illustrated in ❷ of Fig. 2:

1) Bug Descriptions: descriptions detailing the nature of the bug that the patch intends to resolve;
2) Execution Traces: traces of the buggy program's executions;
3) Failing Test Cases: test cases that expose failures in the buggy program;
4) Test Coverage: line and condition coverage metrics for all available test cases associated with the bug.

Bug descriptions, execution traces, and failing test cases serve to enhance LLM4PatchCorrect's comprehension of the characteristics of the bug targeted by a patch generated through a new APR tool. Test coverage serves as an approximate indicator of the adequacy of the available test cases. In cases where test coverage is notably low, even if a patch enables the program to pass all test cases, the correctness of the patch cannot be guaranteed because many code lines and conditions remain uncovered in the tests. By leveraging the diverse range of guiding information (including the labeled patches of existing APR tools), LLM4PatchCorrect can provide accurate predictions.

We evaluate LLM4PatchCorrect using real-world, large-scale patch correctness datasets [23], [29]. These datasets comprise patches generated by 22 different APR tools, with labels meticulously examined by developers. The experimental results demonstrate that LLM4PatchCorrect significantly improves Accuracy, F1, and AUC scores, increasing them from 10.2% to 32.4%, 6.1% to 24.2%, and 10.1% to 34.2%, on average, respectively, compared to state-of-the-art APCA approaches.

Contributions. The main contributions are as follows:

- This paper underscores the importance of a novel setting for the APCA task, where we assume that no labeled patches are available for a new or unseen APR tool. This setting can better match the initial goal of APCA tasks to reduce the manual labeling effort and can evaluate the ability of approaches to transfer knowledge embedded in the existing labeled data to future unlabeled data.
- To the best of our knowledge, we are the first to introduce advanced LLM in solving the APCA task. We design an LLM-based solution, LLM4PatchCorrect, for this challenging setting (i.e., no labeled patches of the new or unseen APR tool).
- We propose incorporating diverse guiding information to aid LLM4PatchCorrect in decision-making regarding patch correctness. Specifically, LLM4PatchCorrect takes into account bug descriptions, execution traces, failing test

TABLE I
OPEN-SOURCED LARGE LANGUAGE MODELS FOR CODE

Models	Structure	Model Size (Billion)	Max Length (Token)	#Training Data
CodeBERT [41]	Encoder	0.13	512	8M instances
CodeT5 [43]	Enc-Dec	0.22	512	8M instances
CodeParrot [45]	Decoder	1.5	1,024	15B tokens
BLOOM-1.1B [46]	Decoder	1.1	2,048	366B tokens
BLOOM-3B [46]	Decoder	3.0	2,048	366B tokens
BLOOM-7.1B [46]	Decoder	7.1	2,048	366B tokens
BLOOM [46]	Decoder	176	2,048	366B tokens
Starcoder-1B [39]	Decoder	1.0	8,192	1,000B tokens
Starcoder-3B [39]	Decoder	3.0	8,192	1,000B tokens
Starcoder-7B [39]	Decoder	7.0	8,192	1,000B tokens
Starcoder [39]	Decoder	15.5	8,192	1,035B tokens
CodeLlama-7B [47]	Decoder	7.0	16,384	500B tokens
CodeLlama-13B [47]	Decoder	13.0	16,384	500B tokens
CodeLlama-34B [47]	Decoder	34.0	16,384	500B tokens
CodeLlama-70B [47]	Decoder	70.0	16,384	1000B tokens

cases, test coverage, and labeled patches generated by existing APR tools.

II. BACKGROUND

This section provides background information on Large Language Models (Section II-A) and their applications in software engineering tasks (Section II-B).

A. Large Language Models (LLMs) for Code

Large Language Models (LLMs) [37], [40], [41], [42], [43], [44], [45], [46] become popular in Natural Language Processing (NLP) and Software Engineering (SE). CodeBERT [37] is a typical encoder-only model for code that is widely used in SE tasks such as code search and defect prediction. CodeT5 [43] is a typical pre-trained encoder-decoder model for code, which is pre-trained on denoising sequence-to-sequence objectives. Starcoder [39], CodeLlama [47], CodeParrot [45], BLOOM [46] are typical LLMs that use only the Transformer decoder to predict the probability of the next token given the previous tokens. The nature of these models makes them highly useful for generation tasks because text/code is usually written in a left-to-right way.

As shown in Table I, except for the structure differences, these large language models for code are also different in model sizes, the maximum tokens can deal with, and the amount of pre-training data. Specifically, if we compare CodeBERT with Starcoder, CodeBERT can at most deal with a code snippet of 512 tokens while Starcoder can deal with a data instance at most consisting of 8,192 tokens. Besides, Starcoder is pre-trained on a giant pre-training data of over 1,000 billion tokens sourced from 80+ programming languages. However, CodeBERT is pre-trained on the CodeSearchNet [48] dataset of 8 million code snippets or documentation including 6 programming languages. For model sizes, the largest variant of Starcoder is about 120 times larger than CodeBERT.

B. Usages of Large Language Models

1) Fine-Tuning: Fine-tuning a large language model for downstream tasks [37], [41], [43], [49] is a prevalent paradigm in the NLP and SE field. Fine-tuning utilizes the knowledge in language models to achieve better model initialization. Using a pre-trained language model for initialization often produces better results with enough labeled data. To adapt the language models to downstream tasks, fine-tuning trains the model in a supervised way. Specifically, given a dataset that consists of task-specific samples X and corresponding labels Y, fine-tuning aims to find a set of parameters θ for the large language model so that $\theta = \arg\max_\theta P(Y|X,\theta)$.

2) In-Context Learning: Though very effective and easy-to-use, fine-tuning usually requires relatively large labeled downstream task datasets to fine-tune *all parameters* in large language models [50]. Besides, fine-tuning demands large GPU resources to load and update all parameters of large language models [51].

An alternative popular approach proposed in GPT3 is in-context learning (ICL) [38], which induces a model to perform a downstream task by inputting examples to the model without any parameter update or training. ICL requires no gradient-based training and therefore allows a single model to immediately perform evaluations on different datasets. ICL mainly relies on the capabilities and knowledge that a large language model learned during its pre-training. The in-context learning makes predictions based on the probability of generating the next token y given the unlabeled data instance x and the context C, which includes k labeled examples. ICL outputs the token y with the highest probability as the prediction for the unlabeled input data x. It can be expressed as: $y = \arg\max_y P_{PTM}(y|C,x)$, where PTM denotes the large language model. C is a context/demonstration created by concatenating k instances along with their corresponding labels i.e., $C = x_1, y_1, x_2, y_2, ..., x_k, y_k$. As shown in the illustration of Fig. 1, ICL asks the large language model to predict the correctness of a test patch given several labeled patches as the context, and the large language model outputs the token "correct" because its probability as the next token is the highest.

III. PROPOSED APPROACH

LLM4PatchCorrect is proposed to utilize labeled patches of existing APR tools to predict the correctness of the patches generated by a new/unseen APR tool. Hereafter, we denote a new or unseen APR tool as the ***target APR tool***. The framework of LLM4PatchCorrect is presented in Fig. 2. It accepts a patch generated by a target APR tool as the raw input (depicted in ❶ of Fig. 2) and produces its correctness label (i.e., correct or overfitting). Specifically, LLM4PatchCorrect consists of the following four main steps:

- **Step 1: Prepare Test Patches** (❶ of Fig. 2). Initially, we gather the patches generated by the target APR tool to form the test set. For each patch within this test set, we append a text prompt to it. Subsequently, we convert the "text prompt + patch" combination into sub-tokens using the tokenizers of LLMs.
- **Step 2: Obtain Similar Patches From Training Set** (❷ of Fig. 2). For each test patch in the test set, we utilize a contrastive learning-based retrieval module to retrieve

several patches with high semantic similarity from the training set for each test patch.

- **Step 3: Obtain Other Guiding Information** (❷ of Fig. 2). For each test patch in the test set, we extract the bug ID targeted by the test patch. We then query the bug benchmark to acquire relevant information about the bug, including bug descriptions, execution traces, failing test cases, and test coverage.
- **Step 4: LLM Inference** (❸ of Fig. 2). We feed the test patch with all the guiding information into the LLM for code (Starcoder-7B). The LLM then predicts the next token conditioning on the input. The predicted next token is then mapped to the patch correctness label.

In the following subsections, we will detail the module for preparing the test patch (Section III-A), the module for retrieving similar patches from the training set (Section III-B), the module for utilizing diverse guiding information (Section III-C), and the module to conduct the LLM inference (Section III-D).

A. Test Patch Preparation

Forming Training/Test Sets. In the upper part of ❶ of Fig. 2, when employing a new and advanced APR tool to address a detected software bug, it may generate many candidate patches capable of passing all available test cases for the identified bug. Subsequently, developers must determine which candidate patch is truly correct and should be implemented. In this study, our goal is to utilize labeled patches from existing APR tools instead of asking developers to manually label those generated by a new or unseen APR tool. In line with this objective, we designate the labeled patches from existing APR tools as the ***training set***, and we designate the patches generated by new or unseen APR tools as the ***test set***. Moreover, we refer to the new or unseen APR tool as the ***target APR tool***.

Prompting Test Patches. For each test patch, we pre-process it based on the recent advances in NLP, namely prompting [38], which can help to better adapt a generic pre-trained model to a specific downstream task. The intuition of prompting is to convert the downstream tasks into a similar form as the pre-training stage. For pre-trained models whose pre-training objective is to predict the next token given previous tokens, e.g., GPT-3 [38] and Starcoder [39], prompting aims to ask a model to predict the next token given previous tokens (patch contents and demonstrations). To help large language models understand task-specific information, prompting modifies the input data by adding a piece of text description, namely prompt templates. We utilize the following prompt template for the test patches:

$\{test\text{-}patch\}$ Q: It was wrong or correct? A: It was

The $\{test\text{-}patch\}$ placeholder is replaced with the test patch content. The large language model (LLM) receives the test patch after patching as input and predicts the next token, indicating the label of the test patch. The process is also presented in ❶ of Fig. 2).

Label Tokens. In this task, we classify patches into two categories: correct patches and overfitting patches. The prompting technique requires two label tokens to distinguish between the two classes (correct and overfitting). Specifically, we use the label tokens "correct" and "wrong" in this work. The label token "correct" is used to indicate correct patches. Additionally, we selected the label token "wrong" to represent the overfitting class. We do not choose the term "overfitting" as a label token because the term "overfitting" has a specific meaning in the APCA task (referring to incorrect patches) and its interpretation can vary in general English. To minimize potential confusion for LLMs, we chose the label token "wrong," a term that is universally understood in both textual and coding contexts, as the label token for the overfitting class in the prompt.

Tokenization. To tokenize the inputs for the large language models, we use their corresponding tokenizers. They are generally built based on the byte pair encoding (BPE) [52], which outputs a sequence of sub-token sequences. BPE can reduce the size of the vocabulary by breaking uncommon long tokens into sub-tokens that frequently appear in the pre-training corpus. Besides, BPE is known to help mitigate the Out-of-Vocabulary (OoV) issue [53].

B. Obtaining Similar Patches From Training Set

1) Motivation: Fine-tuning a large language model such as Starcoder-7B demands extensive computational resources. Hence, we opt to employ In-context learning (ICL) to adapt the LLM to the APCA task. A standard ICL approach [38] treats all the labeled training patches equally and randomly samples k labeled patches to form a demonstration for the LLM, providing the context information about a downstream task. In the APCA task, however, every patch does not equally contribute. For instance, a patch that aims to fix a similar bug to the test patch is more instructive than a randomly sampled one. In other words, we need to design a retrieval module that can select the most valuable labeled patches for each test patch and inspire the LLM to achieve better prediction results.

To select patches, a simple-yet-effective idea is to choose semantically similar patches from the labeled training patches. The semantically similar patches contain similar code changes to the test patch. The difference between similar patches and test patches can possibly contribute to the difference in labels. We assume that providing such semantically similar patch-label pairs to the LLMs can inspire them to learn the context information related to the test patch.

2) Approach: The process of obtaining similar patches from the training set is illustrated in Fig. 3. It involves the following four steps:

1) We first embed patches in both the training and test set into vector representations by utilizing a patch embedding model based on contrastive learning.
2) For each test patch x, we retrieve its top-k most similar patches (i.e., $x_1, x_2, ..., x_k$) among the labeled training patches measuring the distances in the vector space by calculating cosine similarity.
3) The top-k most semantically similar patches are modified via prompting (Section III-A) and then concatenated to form the context/demonstration. The top portion of Fig. 7 illustrates an example of the demonstration comprising similar patches from the training set.

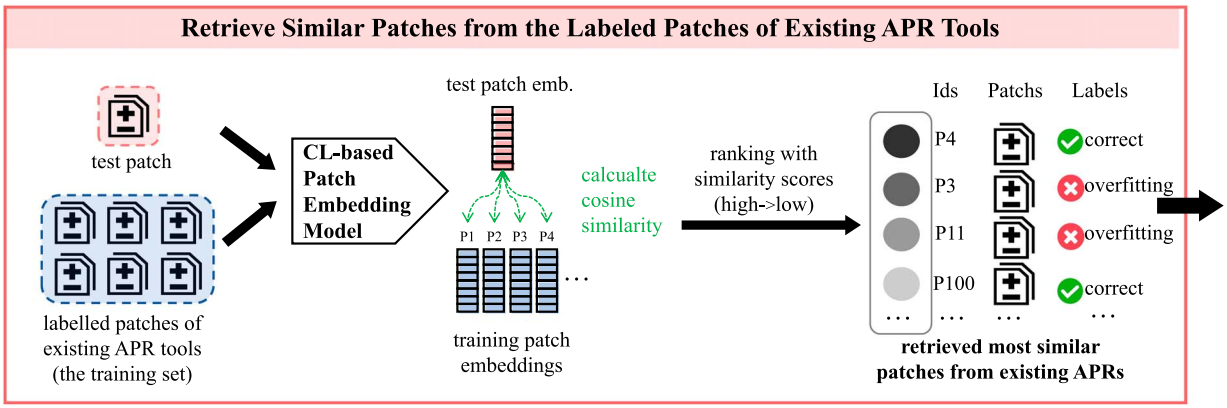


Fig. 3. The process of obtaining similar patches from the training set.

4) The demonstration, along with other guiding information (introduced in Section III-C), is subsequently appended to the test patch x and ultimately fed into the LLM.

Next, we introduce the construction of the contrastive learning-based patch embedding model, which serves as the core of the approach design and is utilized in Step 1 above.

3) Contrastive Learning-Based Embedding Model: It is of great significance to represent patches in suitable embeddings because it will particularly affect whether the patches with the highest similarity scores are semantically similar to the test patch or not. To embed patches into embeddings of good quality, we train an unsupervised patch representation model by leveraging contrastive learning [54].

Triplet Data Construction. In the contrastive learning framework, we need a pre-training dataset whose input instance is in the form of a triplet $\langle p, p^+, p^- \rangle$, where p is a patch, p^+ is a semantically similar patch to p, and p^- is a semantically dissimilar patch to p. Thus, $\langle p, p^+ \rangle$ is considered as a similar pair and $\langle p, p^- \rangle$ is a dissimilar pair. The main training objective of contrastive representation learning is to learn such an embedding space in which similar patch pairs stay closer to each other while dissimilar ones push out far away from each other.

As shown in Algorithm 1, we construct the embeddings of triplets $\langle p, p^+, p^- \rangle$ from the ManySStuBs4J [55] dataset, which contains 153,652 single-statement bug-fix patches mined from 1,000 popular open-source Java projects. This dataset is widely used in many SE downstream tasks such as automated program repair [56], bug detection [57], and fault localization [58]. For each patch p in the ManySStuBs4J dataset, we perform the following steps to generate the embedding of a triplet $t_i = \langle emb(p_i), emb(p_i^+), emb(p_i^-) \rangle$:

- $emb(p_i)$: get one patch p_i from the ManySStuBs4J dataset and feed p_i to the CodeBERT and the first dropout module to get the embedding;
- $emb(p_i^-)$: sample another patch p_j distinct from p_i from the ManySStuBs4J dataset and feed p_j to the CodeBERT and the first dropout module;
- $emb(p_i^+)$: obtain the embedding of the patch resulting from a semantic-preserving transformation applied to p_i (Algorithm 2).

Algorithm 1: Triplet data construction.

Input:
Patches of the ManySStuBs4J dataset, $P = [p_0, ..., p_n]$;
The CodeBERT model, $CodeBERT(\cdot)$;
The first dropout module, $Dropout_1(\cdot)$;
Output:
Embeddings of Constructed triplets, $T = [t_0, ..., t_n]$ where t_i is the embedding of a triplet $\langle p_i, p_i^+, p_i^- \rangle$ and $t_i = \langle emb(p_i), emb(p_i^+), emb(p_i^-) \rangle$

1: **for** each $i \in [0, n]$ **do**
2: obtain the embedding of p_i: $emb(p_i) = Dropout_1(CodeBERT(p_i))$;
3: sample a different patch p_j as p_i^- where $i \neq j$;
4: obtain the embedding of p_i^-: $emb(p_i^-) = Dropout_1(CodeBERT(p_j))$;
5: $emb(p_i^+)$ = GetEmbeddingPositive (p_i)
6: $t_i = \langle emb(p_i), emb(p_i^+), emb(p_i^-) \rangle$
7: append t_i to the list T
8: **end for**
9: **return** T;

Patches p_i and p_i^- can be easily obtained from the ManySStuBs4J dataset, allowing us to easily obtain their embeddings as well. Thus, our primary concern lies in obtaining the embedding of p_i^+ (Algorithm 2). It's crucial to note the **two criteria for a valid positive sample** p_i^+: 1) The embedding of p_i^+ must differ from that of p_i (i.e., $emb(p_i) \neq emb(p_i^+)$) since the two patches are not identical after the transformation. 2) Despite this difference, the embeddings of p_i^+ and p_i should encapsulate the same semantics.

To obtain the embedding of the positive sample p^+, we apply a semantic-preserving transformation proposed in SIMCSE [54] on patches, which is based on the dropout operation. Dropout [59] is a popular technique where randomly selected neurons are ignored during training to alleviate the overfitting problem of neural networks. As shown in Fig. 4, SIMCSE simply applies the standard dropout operation [59] twice to obtain two different embeddings of the same patch p_i. The *first* dropout

Algorithm 2: GetEmbeddingPositive: get the embedding of the positive sample p_i^+.

Input:
The input patch, p_i;
The CodeBERT model, $CodeBERT(\cdot)$;
The second dropout module, $Dropout_2(\cdot)$;
Output:
Embedding of p_i^+, $emb(p_i^+)$;
1: use the second dropout module and CodeBERT to get the embedding of p_i:
$emb_2(p_i) = Dropout_2(CodeBERT(p_i))$;
2: regard the $emb_2(p_i)$ as $emb(p_i^+)$:
$emb(p_i^+) = emb_2(p_i)$
3: **return** $emb(p_i^+)$;

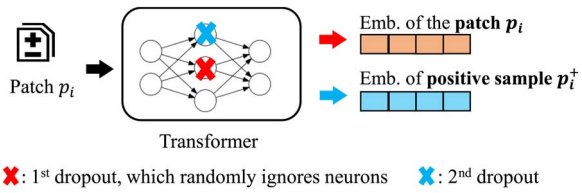


Fig. 4. Generating the embeddings of positive samples via the Dropout operation in SIMCSE. It adopts two different dropout operations which results in two different embeddings.

operation (the red one) randomly ignores a set of neurons, resulting in the embedding of patch p_i, denoted as $emb(p_i)$. Then, SIMCSE adopts the *second* dropout operation (the blue one) which ignores *another set of neurons*, also resulting in a different embedding $emb_2(p_i)$.

The distinction between $emb(p_i)$ and $emb_2(p_i)$ arises from the application of different dropout operations, satisfying the first criterion for a valid p_i^+. Remarkably, despite this difference, both $emb(p_i)$ and $emb_2(p_i)$ originate from the same patch p_i and inherently retain the same semantics, thereby fulfilling the second criterion for a valid p_i^+.

Notably, $emb_2(p_i)$ meets the two criteria required for the embedding of a valid positive sample. Consequently, the authors of SIMCSE regard $emb_2(p_i)$ as the approximation of the embedding of p_i^+ and simply utilize $emb_2(p_i)$ as $emb(p_i^+)$, as shown in Algorithm 2. In essence, SIMCSE does not directly generate a positive patch p_i^+ from p_i; rather, it directly generates the embedding of the patch p_i^+ ($emb(p_i^+) = emb_2(p_i)$) through distinct dropout operations. In other words, SIMCSE applies a transformation at the embedding level, modifying the embeddings, rather than altering the raw data (the patch) itself.

Training Details. After building a large number of triplets from the ManySStuBs4J dataset, we then use those triplets to perform the training of the Contrastive Learning (CL) based patch embedding model. The training details are presented in Fig. 5.

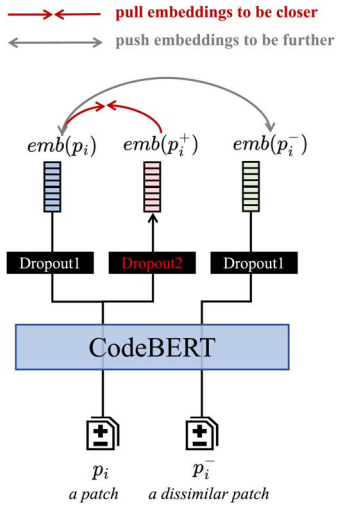


Fig. 5. Training CL-based patch embedding model with the embeddings of the constructed triplet $\langle p, p^+, p^- \rangle$.

Similar to SIMCSE [54], we also use a pre-trained Transformer model as the base model for generating embeddings and add a multi-layer perceptron (i.e., MLP) on top of it. SIMCSE uses RoBERTa [40] as the base model; however, as our data is source code, we use CodeBERT [41], which is pre-trained on both source code and texts. Note that in the framework of contrastive learning, fine-tuning is indispensable to learn such an embedding space in which similar patch pairs stay closer to each other. However, fine-tuning a huge LLM like Starcoder requires vast computation resources that we cannot afford. Thus, we choose to use CodeBERT to learn the contrastive learning-based patch embedding rather than the Starcoder.

As depicted in Fig. 5, a patch p_i and another distinct patch p_i^- from the ManySStuBs4J dataset serve as inputs to the CL-based model during training. We employ the first dropout module (and the CodeBERT) to obtain their respective embeddings $emb(p_i)$ and $emb(p_i^-)$. Following this, we utilize a second dropout module, distinct from the first one, on the patch p_i to acquire the embedding $emb(p_i^+)$. Then, we calculate the loss using the following formula:

$$l_i = -log \frac{e^{cos(emb(p_i),\, emb(p_i^+))/\tau}}{\sum_j^N (e^{cos(emb(p_i),\, emb(p_j^+)/\tau} + e^{cos(emb(p_i),\, emb(p_j^-))/\tau})}$$

where p_i denotes the i-th patch and p_j refers to the j-th patch which iterates over all patches in the mini-batch. N is the number of patches in a mini-batch; τ is a temperature hyper-parameter; and cos is the cosine similarity function. By minimizing the loss above, the model learns such an embedding space in which similar patch pairs stay closer to each other while dissimilar ones push out far away from each other. Then, we can utilize the trained CL-based model to assist in identifying similar patches from the training set when presented with a test patch. For other implementation details, we implement the model by using a popular deep-learning library called HuggingFace[1]. We simply adopt the hyper-parameters recommended by

[1]https://huggingface.co/

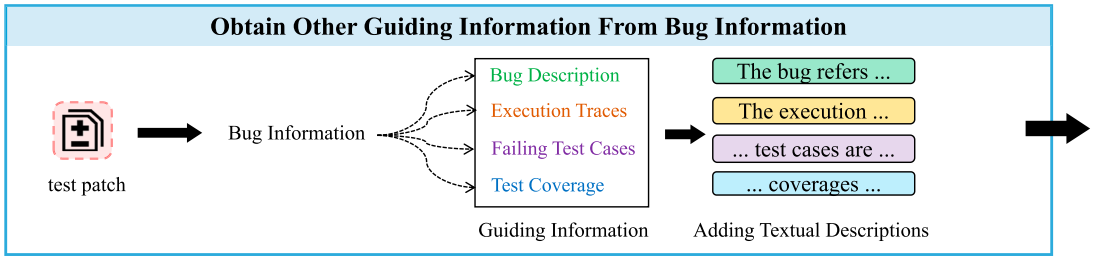


Fig. 6. The process of obtaining the other guiding information from the bug.

SIMCSE [54] which are the learning rate as 5e-5, batch size as 64, and the number of epochs as 3.

C. Obtaining Other Guiding Information

1) Motivation: In addition to the labeled patches of existing APR tools (the training set), LLM4PatchCorrect incorporates a broader range of guiding information, as illustrated in Fig. 6. Bug descriptions, execution traces, and failing test cases contribute to LLM4PatchCorrect's understanding of the bug characteristics addressed by a patch generated through a new APR tool. Test coverage acts as an approximate gauge of the adequacy of the available test cases. In instances where test coverage is notably low, the correctness of a patch cannot be reliably ensured, even if the patch enables the program to pass all test cases, as numerous code lines and conditions remain uncovered.

2) Approach: The process of obtaining the additional guiding information is illustrated in Fig. 6. Firstly, when given a test patch generated by a new APR tool, we will inspect the metadata of the test patch to determine the ID of the bug that the new APR tool is addressing. Secondly, we will use the bug ID to gather the following guiding information:

1) Bug Descriptions: Descriptions detailing the nature of the bug that the patch intends to resolve;
2) Execution Traces: Traces of the buggy program's executions;
3) Failing Test Cases: Test cases that expose failures in the buggy program;
4) Test Coverage: Line and condition coverage metrics for all available test cases associated with the bug.

Lastly, we include the textual description, as depicted in ❷ of Fig. 2, to each piece of information. For instance, for the bug description, we use:

> The bug refers to: [The Description of the Bug]

The placeholder [The Description of the Bug] is replaced with the bug description obtained from the bug benchmark. Similarly, we utilize the textual descriptions from ❷ of Fig. 2 for bug descriptions, execution traces, failing test cases, and test coverage. Bug descriptions, execution traces, failing test cases, and test coverages are details related to the bugs, which can be obtained through program analysis.

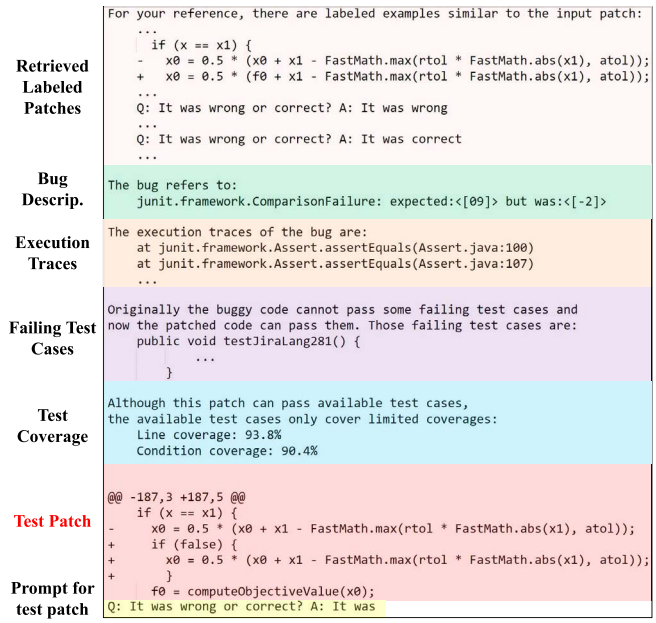


Fig. 7. Example of the concatenated input to the LLM.

D. LLM Inference on Patch Correctness

1) Combining Diverse Guiding Information: LLM4PatchCorrect utilizes a wide range of guiding information. Before conducting the LLM inference, we concatenate each piece of information together and then append the test patch. The concatenation is performed as follows:

$$C = [S_1; S_2; ...; Bug; Trace; Case; Coverage; Test\text{-}Patch]$$

where ";" indicates the concatenation operation. S_j represents the j-th retrieved similar patch from the training set. Please note that each piece of information is prompted with the corresponding textual description, as depicted in ❷ of Fig. 2. For better visualization, Fig. 7 presents one example of the final concatenated input (i.e., C) to the LLM, where the test patch is generated by DynaMoth and aims to address the Math-50 bug in Defects4J.

2) In-Context Learning Inference: The in-context learning inference can be regarded as a text generation problem where the LLM is frozen. Given the concatenated final input C to the LLM, the in-context learning inference outputs the next token y with the highest probability as the prediction for the unlabeled

input data C. It can be formally expressed as:

$$y = \arg\max_{y} \quad P_{LLM}(y|C).$$

where LLM denotes the parameters of the pre-trained model which are frozen all the time. We use a recently proposed LLM, Starcoder-7B [39], the multilingual LLM that is open-sourced, as the backbone model in LLM4PatchCorrect. Starcoder has a list of variants of different model sizes and the largest variant model has 15.5 billion parameters. With the help of the quantization [60] technique, which reduces memory and computational costs by representing weights and activations with lower-precision data types like 8-bit integers (int8), we are able to use any LLM within 7B parameters using a 12GB GPU (the GPU memory of a widely used GPU card 2080-Ti). This indicates we can use any LLM below 7B.

As the patch correctness assessment task is formulated as a binary classification task [23], [33], we first compute the probabilities of the correct class (i.e. $P_{PTM}(correct|C, x)$) and the overfitting class (i.e. $P_{PTM}(overfitting|C, x)$). Specifically, we leverage the LLM's default next-token prediction layer to compute the probabilities. When given an input text/code, the next-token prediction layer returns the probabilities of all tokens in the vocabulary being the next token. Since we use the label tokens "correct" and "wrong" to distinguish between correct patches and overfitting patches (see Section III-A), we calculate the probability of the token "correct" being the next token as $P_{PTM}(correct|C, x)$. Similarly, the probability of the token "wrong" being the next token is $P_{PTM}(overfitting|C, x)$. These probabilities are then normalized between the two classes (correct and overfitting). If the normalized probability of the overfitting class is larger than 0.5, the test patch is predicted as overfitting. Otherwise, it is predicted as correct. We call the probability of the overfitting class generated by a large pre-trained model (after normalization) as the prediction score.

IV. Experimental Setting

In this section, we describe our choice of LLM (Section IV-A), the experimental dataset (Section IV-B), the cross-tool validation setting (Section IV-C), the baselines (Section IV-D), evaluation metrics (Section IV-E), hyper-parameter tuning (Section IV-F), and research questions (Section IV-G).

A. Our LLM Choice

We choose to use Starcoder-7B [39] in this work, as highlighted in Table I. This selection is based on several considerations:

1) CodeBERT and CodeT5 are relatively small in model sizes.
2) CodeParrot and BLOOM have limitations on relatively short input lengths (1,024 and 2,048 tokens, respectively).
3) CodeLlama series are pre-trained on less pretraining code data compared to Starcoder series. Specifically, CodeLlama models (7B, 13B, and 34B) are initialized with Llama 2 model weights and trained on 500 billion tokens from a code-centric dataset [47]. In contrast, the Starcoder series (1B, 3B, and 7B) are trained on 1 trillion tokens from over 80 programming languages [39].
4) Larger models like Starcoder (15.5B) and CodeLlama (13B–70B) require extensive computational resources.

Starcoder-7B strikes a balance between model size and computational requirements, making it suitable for research purposes where computational resources may be limited compared to those available to tech companies. Additionally, its large input range (8,192 tokens) allows us to provide diverse guiding information, such as bug descriptions, execution traces, failing test cases, test coverage, and labeled patches generated by existing APR tools.

B. Dataset

We use the dataset used in Lin et al. [33]'s work containing a total of 1,179 patches from the Defects4J benchmark [61] where most existing APR tools are evaluated. The dataset is merged from two existing large-scale datasets provided by Wang et al. [29] and Tian et al. [23]. Patches in these two datasets were either written by developers (i.e., the ground-truth patches) or generated by 22 different APR tools. Note that their correctness has been carefully labeled and checked.

To ensure the absence of duplicate patches generated by different APR tools, we conducted two verification steps. Firstly, we confirmed the absence of duplicated patches (i.e., identical patches) in the experiment dataset, employing string matching. Secondly, we verified the presence of semantic duplicates—patches that are not identical but possess semantic equivalence—through manual examination of all patches. In this process, we identified two pairs demonstrating semantic equivalence: 1) a pair of patches generated by ACS for the Math-4 bug in Defects4J and 2) a pair of patches generated by ACS for the Lang-35 bug in Defects4J. To address the impact of the semantic equivalent patches, we have removed pairs of semantically duplicate patches from our evaluation data.

C. Cross-Tool Validation

In this paper, we aim to utilize labeled patches of existing APR tools to predict the correctness of the patches generated by a new/unseen APR tool. However, patches generated by future APR tools are impossible to get. Thus, we conduct a "cross-tool" validation that is close to the setting above. In the "cross-tool" scenario, we iteratively regard each APR tool as the target APR tool. For instance, we can first consider the tool `TBar` [11] as the target APR tool, then, all the patches generated by `TBar` are used as the test dataset. At the same time, other patches generated by other APR tools are used as the training data. In this way, we can still evaluate whether the model can transfer the knowledge in labeled patches of APR tools except `TBar` to the patches generated by `TBar`. To carry out the experiment, we employ 22 existing APR tools and construct 22 different sub-datasets in a leave-one-out manner, i.e., we iteratively pick one APR tool as the target APR tool for each sub-dataset. Please note that we remove the patches of the existing APR tools (i.e., the labeled patch pool) that are identical to any patch in the test set to avoid the data leaking issue.

D. Baselines

Please note that each baseline tool is trained on the corresponding training set (i.e., the labeled patches generated by other APR tools) while considering each APR tool as the target at a time. Only LLM4PatchCorrect does not have a training phase, while all the other methods need training. Our baselines are listed as follows:

Patch-Sim [25]. Patch-Sim [25] is a dynamic-based Automated Program Correctness Assessment (APCA) approach that relies on behavior similarities between program executions. It operates under the assumption that passing tests on original and patched programs are likely to behave similarly while failing tests on original and patched programs are likely to behave differently. Based on this observation, Patch-Sim generates new test inputs to enhance the test suites using Randoop. It then utilizes the behavior similarity of these test inputs to determine the correctness of patches.

CodeBERT [41]. Following the success of transformer-based pre-trained models in NLP like BERT [37], researchers have proposed pre-trained models for code, e.g., CodeBERT [41]. CodeBERT is a solid baseline in a wide range of SE downstream tasks such as code search and code summarization. Fine-tuning models typically result in better effectiveness performance compared to freezing models [62]. Additionally, CodeBERT has only 0.13 billion parameters and can be fine-tuned using academic computational resources, such as one 2080-Ti GPU card. Therefore, we directly fine-tune CodeBERT with the training data.

Tian et al.'s Approach [23]. Tian et al. [23] proposed a static-based APCA approach to leverage code (change) representation techniques to predict the correctness (i.e., correct or overfitting) of APR-generated patches. They adopted three recent representation techniques (i.e., BERT [37], CC2vec [63], and Doc2vec [64]) with well-known Machine Learning classifiers (i.e., Logistic Regression, Decision Tree, and Naive Bayes) to demonstrate that it could achieve a promising result. Technically, they froze the representation models and used them to embed patches into distribution embeddings. Then, they fed the embeddings with the labels of patches to Machine Learning classifiers to train classifiers.

ODS [27]. Ye et al. [27] propose ODS, a static-based approach for automated program correctness assessment, which leverages static code features. ODS performs a comparison between a patched program and a buggy program to extract static code features at the AST level. Then, it employs the extracted code features along with patch correctness labels to train a learning-based model.

Quatrain [34]. Tian et al. [34] introduced Quatrain, a static-based approach in Automated Program Correctness Assessment (APCA), which redefines patch correctness evaluation as a question-answering task. Initially, Quatrain employs the natural language processing (NLP) technique to understand the relationship between bug reports and patch descriptions. Subsequently, it constructs a question-answer-based classifier to assess patch correctness.

Cache [33]. Lin et al. [33] proposed an approach named Cache that showed the state-of-the-art performance in the patch correctness assessment task. Cache learns a context-aware code change embedding considering program structures. Specifically, given a patch, Cache focuses on both the changed code and correlated unchanged part and utilizes the AST paths technique for representation where the structure information from the AST node can be captured. After learning the representation, Cache builds a deep learning-based classifier to predict the correctness of the patch.

E. Evaluation Metrics

To evaluate the effectiveness of various target approaches, we adopt widely used evaluation metrics for classification tasks: Accuracy and F1-score. Both Accuracy and F1-score can be measured based on the number of true positives (TP), false positives (FP), and false negatives (FN). Accuracy is defined as the ratio of the number of correctly predicted data (i.e., TP+TN) to the number of all patches (i.e., TP+TN+FP+FN). TP case is referred to when a model prediction is overfitting for an overfitting patch, otherwise, it is an FN case. FP case is referred to when a model prediction is overfitting for a correct patch, otherwise, it is a TN case. F1-score is defined as the harmonic mean of Precision and Recall values. For example, Precision is the ratio of correctly predicted overfitting patches to all the patches predicted as overfitting (i.e., $Precision = \frac{TP}{TP+FP}$) and Recall is the ratio of the number of correctly predicted overfitting patches to the actual number of overfitting patches (i.e. $Recall = \frac{TP}{TP+FN}$). F1-score can be formally defined as $F1\text{-}score = \frac{2 \times Precision \times Recall}{Precision + Recall}$.

As cross-tool validation yields different results for different target APR tools, for easier comparison among methods, we will compute both "averaged results" and "weighted averaged results" across all the APR tools. In the "weighted averaged results", the weights are assigned based on the number of patches in the test sets. The formula for the weighted average is as follows:

$$weighted = \frac{\sum_{i=1}^{22}(\#\text{patch of the tool } i) \times (\text{metric of tool } i)}{\sum_{i=1}^{22} \#\text{patch of the tool } i}$$

where "tool i" refers to the target APR tool and "# patch of the tool i" refers to the number of patches generated by the target APR tool i. "metric of tool i" refers to the metric of a model for the target APR tool i, such as Accuracy and F1-scores.

When calculating the improvement ratios on the (weighted) averaged metrics, we first compute the (weighted) averaged metric across all APR tools. Then, we determine the relative improvement ratios based on the (weighted) average results of the two models. For example, if a patch dataset is generated by three APR tools and a baseline achieves F1-scores of 80%, 70%, and 60% when considering each of the three APR tools as the target tool, and our model achieves F1-scores of 85%, 75%, and 65% accordingly. To compute the relative improvement ratio on F1 based on the average results, we first compute the averaged F1-scores: 70% (i.e., $\frac{80\%+70\%+60\%}{3}$) for the baseline, and 75% (i.e., $\frac{85\%+75\%+65\%)}{3}$) for our model. Then, the relative

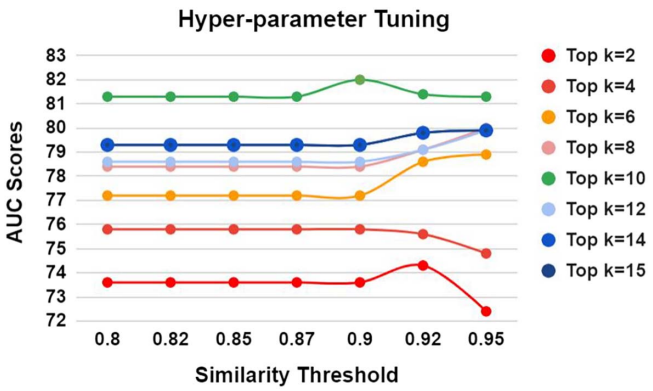


Fig. 8. Hyper-parameter tuning results (AUC) of LLM4PatchCorrect.

improvement ratio of our model over the baseline will be 7.14% (i.e., $\frac{75\%-70\%}{70\%} \times 100\%$).

F. Hyper-Parameter Tuning on Selecting Similar Labeled Patches

As introduced in Section III-B, the final input to LLM4PatchCorrect includes the most similar patches retrieved from the labeled patches of existing APR tools. It is important to decide the way to choose the "most similar patches" given a test patch. The general idea is to include as many similar patches as possible, such that it can provide more valuable information to LLM4PatchCorrect. We specify the general idea into two hyper-parameters:

- the k value: it is the maximum number of the most similar patches that we consider to build the in-context learning demonstration. For instance, "$k = 10$" indicates that we consider up to 10 patches.
- the similarity threshold β: it constrains the similarity of patches for building the demonstration. Specifically, for a test patch, only labeled patches with higher cosine similarities than the threshold β are considered.

To determine a fixed similarity threshold (β) and the maximum number of the most similar patches (k), we conducted preliminary experiments on a randomly split 5% of the labeled patch pools. Specifically, we explored a fine-grained hyper-parameter range: β=[0.80, 0.82, 0.85, 0.87, 0.9, 0.92, 0.95] and k=[2, 4, 6, 8, 10, 12, 14, 15]. As illustrated in Fig. 8, LLM4PatchCorrect exhibits the best performance when k=10 and beta=0.9.

We select the top 10 patches whose cosine similarity scores are higher than 0.9 to a test sample to form the final input to the model.

G. Research Questions

In this paper, we aim at answering the following research questions:

- **RQ1:** How does LLM4PatchCorrect perform compared to state-of-the-art approaches in the cross-tool setting?
- **RQ2:** How does each component of LLM4PatchCorrect contribute?

We design RQ1 to demonstrate the effectiveness of LLM4PatchCorrect by comparing it with state-of-the-art patch correctness assessment approaches. In RQ2, we carefully conduct an ablation study to illustrate the contribution of each component in our solution.

V. Experimental Results

A. RQ1. Overall Performance of the Approach

Main Results. We evaluate the effectiveness of LLM4PatchCorrect by comparing it against the state-of-the-art APCA approaches. As we mentioned in the experimental setting, Tian et al. [23] adopted three representation techniques and three machine learning classifiers. Among the nine variants of the baseline, we report only the best-performing combination to offer readers clearer comparisons of experimental results and more focused insights. For Cache, we reuse the implementation released by the authors of Cache. Table II shows the effectiveness of all the approaches including LLM4PatchCorrect and the baselines in terms of Accuracy, F1-score, and AUC.

The experimental results show that LLM4PatchCorrect outperforms all the baseline techniques. Specifically, LLM4PatchCorrect showed 20.7% 13.0%, and 33.0% improvements against Tian et al.'s work on average in terms of Accuracy, F1-score, and AUC. It also showed 14.6%, 9.2%, and 34.2% of enhancement against Cache on average in terms of Accuracy, F1-score, and AUC. Additionally, LLM4PatchCorrect led to a substantial boost over a strong baseline CodeBERT by 10.2%, 6.1%, and 10.1% in Accuracy, F1-score, and AUC, respectively. Besides, we adopt weighted averages of Accuracy and F1-score where the weights are the number of patches generated by each APR tool. LLM4PatchCorrect still outperforms all baselines. For example, LLM4PatchCorrect leads to 10.0%, 6.8%, and 29.7% improvements over Cache in terms of weighted average Accuracy, F1-score, and AUC. LLM4PatchCorrect outperforms CodeBERT by 9.1%, 5.1%, and 6.1% in terms of weighted average Accuracy, F1-score, and AUC. Furthermore, we conduct the Wilcoxon signed-rank tests between LLM4PatchCorrect and all baselines to investigate whether the improvements are significant. The results show that LLM4PatchCorrect is statistically significantly better than all baselines (all p-values are less than 0.05). Moreover, we observed that the accuracy and F1 scores for SOFix are relatively low compared to other APR tools. This is primarily due to a significant imbalance in the test set, where there are 10 correct patches for every 1 overfitting patch, resulting in a labeling ratio of 1:10. This highly imbalanced test set presents a challenge not only for our approach but also for all methods.

Additionally, it's worth noting that during the execution of ODS, we encountered challenges in performing inference for certain instances when the target APR tools are jKali (4 failing test patches), AVATAR (4 failing test patches), jMutRepair (1 failing test patch), HDRepair (1 failing test patch), and Cardumen (1 failing test patch). Please refer to our replication package for details on all the failing test patches. Consequently,

TABLE II
COMPARISON OF LLM4PATCHCORRECT AND BASELINE METHODS ON THE DEFECTS4J DATASET ALL EVALUATION METRICS ARE HIGHER-THE-BETTER

Defects4J Dataset		Cache [33]			Tian et al. [23]			CodeBERT [41]			ODS [27]			Quatrain [34]			LLM4PatchCorrect		
Target Tools	correct: overfit	Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑
ACS	29:8	27.0	12.9	28.9	40.5	35.3	64.2	43.2	40.0	73.3	45.9	28.6	40.3	37.8	30.3	53.9	**67.6**	**57.1**	**76.7**
Arja	8:49	82.5	89.1	79.6	59.6	72.3	65.6	87.7	93.3	77.6	68.4	78.6	76.5	63.2	74.1	67.1	**94.7**	**97.0**	**89.3**
AVATAR	17:37	74.1	80.6	62.6	74.1	81.6	67.9	75.9	82.7	**83.6**	64.0	71.0	66.1	72.2	80.0	72.2	**85.2**	**90.0**	82.7
CapGen	9:41	84.0	90.2	61.0	84.0	90.0	**94.0**	82.0	90.1	75.9	52.0	62.5	57.5	**86.0**	**91.8**	84.6	80.0	86.8	76.4
Cardumen	0:9	66.7	80.0	-	66.7	80.0	-	77.8	87.5	-	37.5	54.5	-	77.8	87.5	-	**88.9**	**94.1**	-
DynaMoth	1:21	**95.5**	97.6	95.2	90.9	95.2	0.0	**95.5**	**97.7**	81.0	72.7	83.3	**100**	68.2	81.1	28.6	**95.5**	**97.7**	**100**
FixMiner	6:19	72.0	81.1	55.3	68.0	77.8	73.7	80.0	87.2	**87.7**	64.0	71.0	59.6	72.0	82.1	68.4	**84.0**	**89.5**	77.2
GenProg	1:24	88.0	93.6	12.5	68.0	81.0	70.8	**96.0**	**98.0**	50.0	60.0	75.0	0.0	68.0	80.0	**100**	92.0	95.8	16.7
HDRepair	4:4	62.5	72.7	37.5	62.5	66.7	81.2	75.0	80.0	56.2	57.1	66.7	83.3	62.5	72.7	81.3	**87.5**	**85.7**	**100**
Jaid	32:40	61.1	67.4	54.1	61.1	69.6	63.5	**69.4**	**76.1**	70.9	48.6	41.3	44.7	68.1	74.2	70.0	68.1	71.6	**72.6**
jGenProg	6:33	84.6	90.3	**89.4**	56.4	71.2	38.9	87.2	93.0	63.1	69.2	78.6	85.6	82.1	89.6	64.6	**89.7**	**94.3**	82.8
jKali	4:31	85.7	91.5	85.5	68.6	80.7	37.1	**94.3**	96.8	96.8	90.3	94.7	49.1	74.3	84.7	54.8	**94.3**	**96.9**	**99.2**
jMutRepair	2:14	87.5	92.9	50.0	68.8	81.5	46.4	87.5	93.3	82.1	66.7	78.3	63.5	81.3	88.9	64.3	**93.8**	**96.3**	**92.9**
Kali	2:36	89.5	94.4	43.1	76.3	86.6	31.9	**92.1**	**95.9**	**88.9**	84.2	91.4	29.2	71.1	82.5	56.9	**92.1**	**95.9**	77.8
kPAR	2:32	79.4	88.1	57.8	79.4	88.1	54.7	**82.4**	**90.3**	**68.8**	50.0	65.3	64.1	64.7	77.8	62.5	76.5	86.2	56.2
Nopol	6:89	89.5	94.4	42.5	75.8	86.1	38.0	69.5	81.5	60.9	51.6	65.7	**69.5**	71.6	83.2	46.4	**93.7**	**96.7**	54.1
RSRepair	2:31	81.8	89.3	85.5	81.8	89.7	72.6	**90.9**	**95.2**	61.3	75.8	85.2	87.1	75.8	85.7	80.6	**90.9**	95.1	**88.7**
SequenceR	10:45	76.4	**84.7**	61.1	**80.0**	87.1	75.1	61.8	72.0	72.7	74.5	83.3	59.3	67.3	79.1	57.3	72.7	80.0	**78.2**
SimFix	16:42	69.0	78.6	49.0	75.9	82.5	74.1	70.7	80.0	68.9	60.3	70.9	65.8	69.0	79.5	69.2	**77.6**	**85.1**	**79.2**
SketchFix	5:7	50.0	62.5	65.7	66.7	66.7	68.6	50.0	50.0	60.0	83.3	85.7	80.0	33.3	33.3	54.3	**91.7**	**92.3**	**94.3**
SOFix	10:1	27.3	20.0	70.0	54.5	28.6	80.0	27.3	20.0	90.0	**72.7**	**40.0**	95.0	45.5	25.0	80.0	54.5	28.6	**100**
TBar	7:33	85.0	90.6	71.4	77.5	86.2	70.1	**87.5**	**92.8**	63.2	52.5	61.2	80.3	75.0	84.8	65.8	85.0	90.6	**92.4**
Average	-	73.6	79.2	59.9	69.9	76.6	60.4	76.5	81.5	73.0	63.7	69.7	64.6	67.6	74.9	65.8	**84.4**	**86.5**	**80.4**
Improve. over baseline	-	+14.6%	+9.2%	+34.2%	+20.7%	+13.0%	+33.0%	+10.2%	+6.1%	+10.1%	+32.4%	+24.2%	+24.4%	+24.8%	+15.5%	+22.0%	-	-	-
Weighted Average	-	76.4	81.9	59.3	70.9	78.9	59.9	77.0	83.3	72.5	62.3	69.1	62.5	69.4	77.8	64.5	**84.0**	**87.5**	**76.9**
Improve. over baseline	-	+10.0%	+6.8%	+29.7%	+18.5%	+10.9%	+28.3%	+9.1%	+5.1%	+6.1%	+34.9%	+26.7%	+23.0%	+21.1%	+12.4%	+19.2%	-	-	-

Note: The best performer for each metric and target tool is highlighted in blue and bold.

TABLE III
COMPARISON OF LLM4PATCHCORRECT WITH ODS ON APR DATA WHERE ODS INFERENCE FAILS IN SOME PATCHES

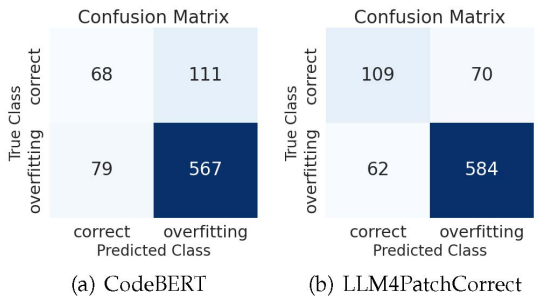
Target APR Tools	correct: overfit	ODS			LLM4PatchCorrect		
		Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑
AVATAR	16:34	64	71	66.1	**86.0**	**90.4**	**84.7**
Cardumen	0:8	37.5	54.5	-	**87.5**	**93.3**	-
HDRepair	4:3	57.1	66.7	83.3	**85.7**	**80.0**	**100**
jKali	4:27	90.3	94.7	49.1	**93.5**	**96.4**	**99.1**
jMutRepair	2:13	66.7	78.3	63.5	**93.3**	**96.0**	**92.3**

TABLE IV
COMPARISON OF LLM4PATCHCORRECT WITH PATCH-SIM ON DATASET FROM [29]

Target APR Tools	Patch-Sim			LLM4PatchCorrect		
	Acc↑	F1↑	AUC↑	Acc↑	F1↑	AUC↑
ACS	**65.0**	22.2	**53.3**	45.0	**52.2**	45.1
Arja	35.1	46.4	55.4	**94.7**	**97.1**	**81.2**
AVATAR	50.0	54.2	54.0	**77.8**	**85.4**	**77.7**
CapGen	61.2	67.5	60.1	**70.1**	**73.7**	**81.4**
Cardumen	41.7	53.3	-	**91.7**	**94.7**	**-**
DynaMoth	31.8	44.4	64.3	**95.5**	**97.7**	**95.2**
FixMiner	50.0	46.7	55.0	**78.1**	**83.7**	**82.5**
GenProg	35.7	50.0	66.7	**96.4**	**98.1**	**100**
Jaid	**63.0**	59.5	**62.9**	56.8	**61.5**	60.9
jGenProg	47.4	61.5	41.7	**84.2**	**90.9**	**66.7**
jKali	45.8	58.1	70.5	**95.8**	**97.7**	**97.7**
jMutRepair	59.1	69.0	59.4	**90.9**	**94.1**	**95.3**
Kali	31.7	43.8	64.0	**95.0**	**97.4**	**78.9**
kPAR	48.3	59.7	53.0	**86.7**	**92.2**	**79.8**
Nopol	33.3	47.4	**65.5**	**96.7**	**98.3**	51.7
RSRepair	30.0	41.7	63.2	**95.0**	**97.4**	**92.1**
SequenceR	49.3	59.1	51.8	**70.4**	**76.9**	**82.2**
SimFix	50.0	50.7	57.0	**77.3**	**84.8**	**77.1**
SketchFix	**70.8**	36.4	58.4	**70.8**	**58.8**	**75.6**
SOFix	**73.9**	**50.0**	85.0	**73.9**	**50.0**	**95.0**
TBar	52.9	53.5	59.3	**82.9**	**88.0**	**86.2**
Average	48.9	51.2	60.0	**82.2**	**84.3**	**80.1**
Weighted Average	49.0	52.8	59.3	**80.5**	**83.9**	**79.4**

the performance metrics of ODS presented in Table II are calculated based on the successfully tested patches only. To ensure a fair comparison, we also evaluate LLM4PatchCorrect using the identical test sets that ODS can successfully infer. As there are differences when the target APR tools are jKali, AVATAR, jMutRepair, HDRepair, and Cardumen, we only re-evaluate LLM4PatchCorrect on those APR tools. Table III displays the performance of ODS and LLM4PatchCorrect on identical test sets. Our approach consistently and significantly outperforms ODS.

Comparison with Patch-Sim. We also conducted a comparison between our LLM4PatchCorrect and the dynamic-based APCA approach Patch-Sim [25]. Because implementing Patch-Sim on new datasets is complex, we relied on prediction results from an empirical study by Wang et al. [29] for comparison. Specifically, we used the dataset from their study to evaluate LLM4PatchCorrect in the cross-tool validation setting and compared Patch-Sim's performance in the same dataset under the same cross-tool validation. As shown in Table IV, our approach significantly outperforms Patch-Sim on average of all the target APR tools. Specifically, LLM4PatchCorrect showed 68.1%, 64.6%, and 33.5% improvements against Patch-Sim on average in terms of Accuracy, F1-score, and AUC.



(a) CodeBERT (b) LLM4PatchCorrect

Fig. 9. Confusion Matrices of LLM4PatchCorrect and the best-performing baseline CodeBERT.

Confusion Matrix Analysis. Fig. 9 compares the confusion matrices of LLM4PatchCorrect and the best-performing baseline (i.e. CodeBERT). For ease of analysis, we put the predictions of each test set in the "cross-tool" setting together and compute the confusion matrices. In total, we have 825 patches generated by 22 APR tools. Please note that though the whole dataset has 1,179 patches, only 825 of them are generated by APR tools and the other patches are contributed by developers. Among the 825 generated patches, 646 patches are overfitting and only 179 patches are correct. The correct patch ratio of APR-generated patches is only 21.7% ($\frac{179}{179+646}$) without using APCA techniques. Using LLM4PatchCorrect, it can filter out 584 overfitting patches (i.e., 90.4% of overfitting patches) as shown in Fig. 9(b). The remaining patches are predicted as correct by LLM4PatchCorrect and the correct patch ratio of the remaining patches is increased from 21.7% to 63.7% ($\frac{109}{109+62}$). For CodeBERT, as shown in Fig. 9(a), it can filter out 567 overfitting patches (i.e., 87.7% of overfitting patches) but the correct patch ratio of the remaining patches is only 46.3% ($\frac{68}{68+79}$). LLM4PatchCorrect has a 37.6% ($\frac{63.7\%-46.3\%}{46.3\%}$) relative improvement over CodeBERT in terms of the correct patch ratio of the remaining patches. Though APCA techniques can help to reduce the number of overfitting patches, they inevitably delete some correct patches at the same time. Fig. 9(a) also shows that CodeBERT wrongly predicts 111 correct patches as "overfitting" while the corresponding number for LLM4PatchCorrect is 70. It indicates that LLM4PatchCorrect makes fewer mistakes than CodeBERT.

False Positive and False Negative Rates. False Positive Rate (FPR) indicates how often a model incorrectly predicts an instance as positive (an overfitting patch) when it is actually negative (a correct patch). False Negative Rate (FNR) indicates how often a model incorrectly predicts an instance as negative (a correct patch) when it is actually positive (an overfitting patch). Lower values for both FPR and FNR indicate better performance, with the ideal model minimizing both metrics. Table V presents the average FPR, FNR, and their sum for our tool and the baseline models. Our tool achieves the second-best FPR, the best FNR, and the lowest combined FPR and FNR score. While ODS slightly outperforms our tool in FPR, it has a substantially higher FNR. This suggests that although ODS classifies fewer correct patches as overfitting, it does so at the cost of incorrectly labeling many more overfitting patches as correct. This misclassification can lead to increased manual effort, as users may spend more time dealing with overfitting patches that are mistakenly predicted as correct and subsequently adopted. In contrast, our tool provides a more balanced approach by achieving strong performance in both FPR and FNR.

TABLE V
AVERAGE FALSE POSITIVE AND FALSE NEGATIVE RATES OF LLM4PATCHCORRECT AND BASELINES ON THE DEFECTS4J DATASET

Average	Cache	Tian et al.	CBERT	ODS	Quatrain	LLM4PatchCorrect
FPR↓	49.2	58.7	67.7	**38.6**	56.5	42.8
FNR↓	17.4	22.1	11.1	33.3	22.6	**9.6**
FPR+FNR↓	66.6	80.8	78.8	71.9	79.1	**52.4**

CBERT: CodeBERT [41]

TABLE VI
RESULTS OF THE ABLATION STUDY IN TERMS OF ACCURACY, F1-SCORE, AND AUC, ON AVERAGE OF ALL APR TOOLS

Ablation	LLM	bug descri.	exe. traces	test cases	test cover.	retrieved patches	Acc.	F1	AUC
No Design	●						75.4	83.1	41.8
Info.of	●	●	●				75.7	83.5	45.8
Bug/Tests	●			●	●		76.0	83.7	43.3
Labeled Patches	●					●	84.0	86.3	77.2
Full	●	●	●	●	●	●	**84.4**	**86.5**	**80.4**

Answer to RQ1: LLM4PatchCorrect significantly outperforms the baseline techniques. By utilizing both the labeled patches from existing APR tools and additional guiding information from the bug benchmark, our approach accurately predicts the correctness of patches generated by the target APR tool, even in the absence of labeled patches specific to that tool.

B. RQ2. Ablation Study

Method. To delineate the contributions of each component of LLM4PatchCorrect, we conduct an ablation study. The ablation studies are based on three groups of guiding information:

- Bug Information: Bug descriptions and execution traces;
- Test Information: Failing test cases and test coverages;
- Retrieved Patches: Patches retrieved from the existing APR tools (the training set).

For the ablation study, we initially employed only the LLM (Starcoder-7B) without any guiding information. Subsequently, we combined each category of the three groups of guiding information with the LLM (Starcoder-7B) to perform the APCA task. Finally, we incorporate all the guiding information (the full model) to complete the task.

Results. We initially investigated the effects of integrating guiding information from the bug benchmark. As depicted in Table VI, on average, utilizing bug-related information, including bug descriptions and execution traces, consistently yields

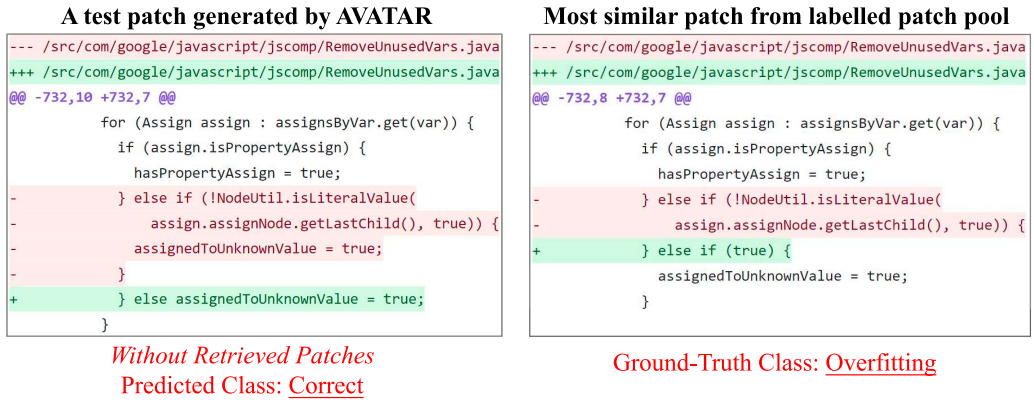


Fig. 10. A test patch generated by AVATAR and the predictions given by LLM4PatchCorrect with and without the similar patch from the training data.

improvements across all metrics. These enhancements demonstrate a relative improvement of up to 9.6% compared to solely relying on the large language model (e.g., Starcoder-7B). Additionally, information related to test cases (i.e., failing test cases and test coverages) also consistently leads to improvements across all metrics compared to relying solely on the large language model. Additionally, we examined the contribution of the contrastive learning-based patch retrieval module, which retrieves patches from the training set. As shown in Table VI, on average, the patch retrieval module leads to relative improvements of 11.4%, 3.9%, and 84.7% in Accuracy, F1-score, and AUC, respectively, compared to relying solely on the large language model. Moreover, we discovered that the guiding information from the bug benchmark and the retrieved patches complement each other, resulting in improved results for the full model of LLM4PatchCorrect.

Answer to RQ2: All modules contribute to the effectiveness of LLM4PatchCorrect. Incorporating all designs, our approach achieves relative improvements of 11.9%, 4.1%, and 92.3% in Accuracy, F1-score, and AUC, respectively, compared to relying solely on the large language model.

VI. Discussion

A. Case Study

The ablation study (RQ2) revealed that the contrastive learning-based patch retrieval module, utilized to acquire semantically similar patches from the training set, plays the most important role in ensuring the accurate prediction of LLM4PatchCorrect. In this discussion, we give a case study to demonstrate how our proposed contrastive learning-based patch retrieval module would enhance the LLM4PatchCorrect. To achieve this, we compare the prediction made by the complete LLM4PatchCorrect model with that of its variant that excludes the retrieved labeled patches when evaluating the correctness of a given test patch.

Fig. 10 illustrates an example of a test patch extracted from the `closure-compiler` project, aiming to fix the Closure-45 bug in Defects4J, along with the most similar patch retrieved from the training set. Such a test patch (the left-hand side code) tries to repair the program by removing the condition of the `else if` statement. LLM4PatchCorrect without retrieved labeled patches predicts the input test patch as correct. But this test patch, in fact, is an overfitting patch. However, using the full LLM4PatchCorrect (including the retrieved labeled patches) the prediction is changed to overfitting which is the ground truth. The most similar patch retrieved from the training set (the right-hand code) also tries to repair the program by relaxing the condition of the `else if` statement but this patch is already identified as an overfitting patch. Since the two patches are similar to each other, the label of the similar patch is valuable for the LLM in assessing the test patch. Moreover, please kindly note that the patch on the right in Fig. 10 is retrieved from the training set by our approach, but it is also utilized in the training of all baselines like Cache. In other words, this patch on the right is "seen" by all baselines during their model training, ensuring fairness in the evaluation of baselines.

B. Effectiveness on the Bears Benchmark

The main experiments are conducted based on the patches for Defects4J benchmark [61]. To ensure that our approach can work beyond the Defects4J benchmark, we did additional experiments on the Bears benchmark [65].

The patches used in this discussion are sourced from [27] and we include those patches generated by four APR tools: Arja [66], along with GenProg [67], Kali [68], and RSRepair [69], which were reimplemented by the authors of Arja [66] to support Java. We only consider the patches generated by the four tools above because only those patches are supported by the implementations of all baselines. To conduct the experiments, we still follow our cross-tool validation setting, where we consider the patches generated by one APR as the test set and the rest as the training set. Tables VII and VIII demonstrate the effectiveness of both the baselines and our approach in the Bears benchmark. The experimental results highlight the exceptional performance of LLM4PatchCorrect in the Bears

TABLE VII
ACCURACY OF LLM4PATCHCORRECT AND BASELINES ON THE BEARS BENCHMARK [65]

Accuracy	Tain et al.	CodeBERT	ODS	Quatrain	Cache	Ours
Arja	9.0	4.3	3.7	7.3	3.7	**85.0**
GenProg	73.4	88.3	47.8	99.5	**100**	**100**
Kali	16.7	16.7	0.0	**83.3**	40.0	**83.3**
RSRepair	**100**	**100**	**100**	**100**	**100**	**100**
Average	49.8	52.3	37.9	72.5	60.9	**92.1**
Weighted Average	36.4	39.9	22.5	47.1	44.7	**91.3**

TABLE VIII
F1-SCORE OF LLM4PATCHCORRECT AND BASELINES ON THE BEARS BENCHMARK [65]

F1	Tain et al.	CodeBERT	ODS	Quatrain	Cache	Ours
Arja	16.5	8.3	7.1	13.6	7.1	**91.8**
GenProg	84.7	93.8	64.6	99.8	**100**	**100**
Kali	28.6	28.6	0	90.9	57.1	**90.9**
RSRepair	**100**	**100**	**100**	**100**	**100**	**100**
Average	57.5	57.7	42.9	76.1	66.1	**95.7**
Weighted Average	45.5	44.6	31.4	50.8	46.9	**95.2**

benchmark. On average, LLM4PatchCorrect outperformed the best-performing baseline (Quatrain [34]) by 27.0% ($\frac{92.1-72.5}{72.5}$) in Accuracy and 25.8% ($\frac{95.7-76.1}{76.1}$) in F1-score on average. This underscores the effectiveness of LLM4PatchCorrect in a new dataset beyond the Defects4J.

C. Impact of the LLM Choices

In this subsection, we aim to discuss the impacts of selecting different LLM models as the backbone model. Prior research [37], [38], [70] suggests that increasing the size of LLMs, such as expanding the pre-training corpus and model parameters, generally leads to improved performance. Due to computational resource constraints in academic settings, we limit our selection to LLMs up to 7B parameters. We choose the LLMs up to 7B and we argue that the 7B parameters are sufficiently large for research purposes.

In addition to the Starcoder-7B model, we also implemented a version of LLM4PatchCorrect based on another popular 7B model, named Code LLamma-7B [47]. Furthermore, to validate the relationship between model sizes and effectiveness within the framework of LLM4PatchCorrect, we included smaller LLMs such as CodeGen2-3.7B [71], Starcoder-3B [39], Starcoder-1B [39], and BLOOM-1.7B [46]. The results are presented in Table IX, with the highlighted row indicating the original version of LLM4PatchCorrect. As shown in Table IX, we observe that larger model sizes generally exhibit better performance. Nevertheless, we note that Starcoder-1B also exhibits competitive performance despite its relatively small model size. Moreover, we find that the effectiveness of LLM4PatchCorrect remains stable when replacing Starcoder-7B with another popular 7B model; the difference in effectiveness is less than 1%, indicating that LLM4PatchCorrect maintains satisfactory performance with a different LLM.

TABLE IX
IMPACTS OF DIFFERENT LLMS ON LLM4PATCHCORRECT IN THE EVALUATION DATASET BASED ON DEFECTS4J

Model Size (M)	LLMs	Acc.	F1	AUC
M=7B	Starcoder-7B [39] (Ours)	**84.4**	**86.5**	**80.4**
	CodeLLama-7B [47]	83.7	85.6	**80.4**
2B<M<7B	CodeGen2-3.7B [71]	81.3	84.1	77.6
	Starcoder-3B [39]	80.1	82.6	76.6
1B<M<2B	BLOOM-1.7B [46]	80.4	83.2	76.4
	Starcoder-1B [39]	81.9	85.9	73.8

D. Application Scenario

When a user applies automated program repair tools to fix a bug, the tools may generate multiple plausible patches that pass all available test cases. However, manually determining which of these patches truly resolves the bug is time-consuming and tedious.

Our method introduces a novel approach to assess patch correctness automatically by leveraging the capabilities of Large Language Models (LLMs). Developers can benefit from LLM4PatchCorrect on accurate assessment of which patches truly fix the bug, as it not only harnesses the power of LLMs and labeled patches but also integrates a broad spectrum of relevant bug-related information, such as bug descriptions, execution traces, failing test cases, and test coverage. By supporting the inclusion of comprehensive bug-related details, our method allows users to input information that may have been underutilized in previous studies. With LLM4PatchCorrect, users can more quickly and confidently select appropriate bug-fixing solutions, thereby enhancing overall repair efficiency and software quality.

E. Limitations

Our approach, like many other learning-based static patch correctness assessment approaches [23], [27], [33], relies on the availability of labeled patches from either other APR tools or developers as training data. Labeled training data is needed for learning-based approaches to understand the patterns of correct and overfitting patches. Therefore, labeled training data is essential for all learning-based static patch correctness assessment approaches, not solely for our approach. One of the ultimate goals of this line of work is to diminish the reliance on labeled data while ensuring high accuracy and efficiency. Our work has made an exploration to remove the need for labeling the patches generated by new/unseen APR tools while maintaining the high accuracy of predictions, which pushes the research on static patch correctness assessment approaches toward the goal of removing the need for labeled data. However, at this stage, our approach still depends on labeled patches from existing APR tools, which presents a limitation that future studies need to address.

In addition, our approach has a limitation in training the retriever, as we only utilized single-line patches in training the retriever through contrastive learning (Section III-B). Contrastive learning helps the base model better understand patch

similarity. However, the evaluation datasets of the patch correctness assessment task contain multi-line patches, which may hinder the retriever from achieving optimal effectiveness on these evaluation datasets. Even though the retriever is trained on single-line patches, it still acquires valuable knowledge for identifying similar patches, in contrast to models not trained with contrastive learning. In the future, we plan to incorporate multi-line patches into the training data for the retriever, aiming to enhance the retriever's generalizability to multi-line patches.

F. Threats to Validity

Threats to External Validity. Large pre-trained models diverge depending on different aspects, such as the characteristics of pre-training tasks and the size of pre-training datasets. As a threat to validity, our study may have a selection bias by considering only several large pre-trained models. To mitigate this threat, we conducted preliminary experiments with existing open-source models and kept tracking the models that are not publicly shared online. Another threat to validity can be dataset selection, as it may deliver bias in the experimental results. The selection, however, is to compare against the state-of-the-art following their settings. Researchers [23], [29] reported that the dataset had been checked for its correctness which automatically minimized this threat to validity. The evaluation metrics we borrow sometimes may cause bias depending upon the characteristics of the tasks. We believe this threat is mitigated as we double-checked, and they are well-known for classification tasks. Furthermore, We publicly share our implementation and dataset for future comparisons by the research community. Finally, we acknowledge another threat to validity: both the training and test data are based on bugs from a restricted set of software, such as Defects4J and Bears. Our tool may not generalize effectively to a wider range of software applications, potentially affecting its performance. However, this is a common limitation across all studies in this task, and we consider addressing this challenge an important direction for future work.

Threats to Internal Validity. The main threat to internal validity lies in the manually crafted prompt that we designed for our model. We cannot ensure that our prompt is optimal as well as it is impossible to traverse all the potential prompts. We mitigate this by following the most common prompts [62] and we share the prompt in the artifacts for the community to review.

Threats to Construct Validity. The large pre-trained model we employ in our study is not perfect, and it may have been under-trained, which can affect its complete effectiveness as we pre-envisioned in the previous Section. This may imply that our design can boost the tool's effectiveness by capturing more practical features for the assessment. We believe our future study can shed light on this threat by considering larger models.

VII. Related Work

Many automated patch correctness assessment (APCA) approaches [4], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35] have been proposed to conduct the patch correctness assessment. The APCA approaches can be categorized into two categories: (1) dynamic approaches which are based on running/executing the tests and (2) static approaches which are built on top of source code patterns or features.

Dynamic APCA approaches. Yang et al. [26] leveraged fuzz strategies on existing test cases to automatically generate new test inputs. Xin and Reiss [30] utilized the syntactic differences between the buggy code and its patched code to generate new test inputs. Xiong et al. [25] focus on the behavior similarity of the failing tests on buggy and patched programs to assess the correctness of generated patches. Ye et al. [28] utilized a technique called "Random testing with Ground Truth (RGT)" [72] to generate extra tests based on the human-written patch.

While dynamic approaches have shown promise, they tend to be highly time-consuming [28], [30], whereas static approaches offer greater efficiency. Our approach falls within the category of static approaches. In this study, we aim to advance static APCA approaches. We consider both static and dynamic approaches to be valuable research directions that contribute to progress in the field.

Static APCA approaches. Ye et al. [27] proposed ODS to detect overfitting patches. They first statically extracted 4,199 code features at the AST level from the buggy code and generated patches by the APR tools. Then they fed the extracted features to three machine learning algorithms (Logistic Regression, k-Nearest Neighbors, and Random Forest) and ensemble the three models to assess the correctness. Tian et al. [23] leveraged representation learning techniques (e.g., BERT [37]) to build embeddings for overfitting and correct patches generated by APR tools. They then fed the embeddings to machine learning classifiers (e.g. Logistic Regression) to obtain prediction results. Phung et al. [73] propose MIPI, an approach that leverages the similarity between the patched method's name (often indicative of the developer's intention) and the semantic meaning of the method's body (reflecting the implemented behavior) to detect and eliminate overfitting patches generated by APR tools. Ghanbari and Marcus [74] proposed Shibboleth, a method that measures the impact of patches on both production code and test code to separate the patches that result in similar programs. Shibboleth assesses the correctness of patches via both ranking and classification.

Recently, Tian et al. [34] introduced Quatrain, a static-based approach that redefines patch correctness evaluation as a question-answering task. Quatrain employs the natural language processing (NLP) technique to understand the relationship between bug reports and patch descriptions. Lin et al. [33] proposed Cache that utilized both the context and structure information in patches. Cache achieved state-of-the-art performances in the APCA task by outperforming existing dynamic and static APCA tools. manually label the patches generated by a new/unseen APR tool, which indeed alleviates the manual labeling process for the APCA task.

While LLM4PatchCorrect utilizes the same labeled patches in the training set as other learning-based approaches, it distinguishes itself in the way it uses these patches. Previous methods like Cache and Quatrain [33], [34] incorporate these patches during model training, whereas our approach employs them

during inference. We adopt this strategy due to the challenge of updating the parameters of large language models (LLMs), thus resorting to in-context learning to guide LLMs. Despite this difference in approach, our method doesn't necessitate additional labeled data or a correctness oracle. Hence, our comparison with baselines remains fair.

Other Studies. Recently, Le-Cong et al. [35] proposed Invalidator that utilized both dynamic features (i.e., program invariants) and static features (i.e., code embedding extracted from CodeBERT). However, it is time-consuming to generate the dynamic features. Invalidator took five hours to infer dynamic features and seven minutes (on average) to assess the correctness for a single patch. While LLM4PatchCorrect only costs 2.4 seconds for each patch. Wang et al. [29] performed a large-scale empirical study on the effectiveness of existing APCA approaches. Yang et al. [75] constructed a new APCA dataset and conducted an empirical study on the effectiveness of existing APCA approaches on the new dataset. Motwani et al. [76] investigate the phenomenon in real-world Java programs, assessing the effectiveness of four program repair tools (GenProg, Par, SimFix, and TrpAutoRepair) on defects introduced by the projects' developers during their regular development process. They discovered that the generated patches frequently overfit to the provided test suite, with only 13.8% to 46.1% of the patches passing an independent set of tests. Additionally, Xia et al. [77] introduced AlphaRepair, an infilling-style APR approach for generating bug fixes without the need for fine-tuning pre-trained models. While AlphaRepair shares similarities with our approach in not requiring fine-tuning, our focuses differ: Xia et al. concentrate on the program repair task, whereas we center our attention on patch correctness assessment. Furthermore, while we utilize large-size LLMs up to 7B parameters, Xia et al. only employ the small-size LLM CodeBERT with 0.13B parameters.

Lastly, Ye et al. [78] applied three patch correctness assessment techniques based on random testing, test generation, and invariant detection to comprehensively study the presence of overfitting patches in QuixBugs [79]. In contrast, our work mainly focuses on the Defects4J [61] and Bears [65] benchmarks with the use of LLMs.

VIII. Conclusion and Future Work

In this study, we introduce LLM4PatchCorrect, the first Large Language Models (LLMs) based automatic patch correctness assessment technique. LLM4PatchCorrect utilizes an advanced LLM for code (Starcoder-7B) to assess the correctness of unlabeled patches of a new or unseen APR tool, without the need for fine-tuning. Moreover, LLM4PatchCorrect incorporates a contrastive learning-based retrieval module to select similar patches as examples for each test patch, which helps the LLM better understand the correctness of the test patch. Additionally, LLM4PatchCorrect integrates diverse guiding information to enhance its decision-making process. Specifically, LLM4PatchCorrect incorporates bug descriptions, execution traces, failing test cases, and test coverage data. Our experimental results showed that LLM4PatchCorrect can achieve an accuracy of 84.4% and an F1-score of 86.5% on average although no labeled patch of the new or unseen APR tool is available. In addition, LLM4PatchCorrect significantly improves Accuracy, F1, and AUC scores, increasing them from 10.2% to 32.4%, 6.1% to 24.2%, and 10.1% to 34.2%, on average, respectively, compared to state-of-the-art approaches. For future work, we would like to explore the effectiveness of LLM4PatchCorrect on other tasks such as just-in-time defect prediction.

Data Availability

To facilitate future studies, our implementation of LLM4PatchCorrect and experimental data are publicly available at: https://github.com/Xin-Zhou-smu/LLM4PatchCorrectness.

Acknowledgment

This research / project is supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, pp. 54–72, 2012.
[2] F. Long and M. C. Rinard, "Staged program repair with condition synthesis," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 166–178.
[3] X.-B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, 2016, pp. 213–224.
[4] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2017, pp. 660–670.
[5] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.
[6] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "Lsrepair: Live search of fix ingredients for automated program repair," in *Proc. 25th Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2018, pp. 658–662.
[7] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2018, pp. 118–129.
[8] B. Lin et al., "Understanding the non-repairability factors of automated program repair techniques," in *Proc. 27th Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2020, pp. 71–80.
[9] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "On the impact of flaky tests in automated program repair," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2021, pp. 295–306.
[10] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, 2018.
[11] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 31–42.
[12] Q. Zhu et al., "A syntax-guided edit decoder for neural program repair," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 341–353.
[13] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, 2021, pp. 1161–1173.
[14] F. Long and M. C. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 702–713.

[15] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," *Empirical Softw. Eng.*, vol. 23, pp. 3007–3033, Jun. 2018.

[16] X.-B. D. Le, L. Bao, D. Lo, X. Xia, and S. Li, "On reliability of patch correctness assessment," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 524–535.

[17] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2019, pp. 1–12.

[18] A. Nilizadeh, G. T. Leavens, X.-B. D. Le, C. S. Pasareanu, and D. R. Cok, "Exploring true test overfitting in dynamic automated program repair using formal methods," in *Proc. 14th IEEE Conf. Softw. Testing, Verification Validation (ICST)*, 2021, pp. 229–240.

[19] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 3007–3033, 2018.

[20] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.

[21] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 672–681.

[22] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 165–176.

[23] H. Tian et al., "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Piscatway, NJ, USA: IEEE Press, 2020, pp. 981–992.

[24] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 727–738.

[25] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 789–799.

[26] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 831–841.

[27] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 2920–2938, Aug. 2022.

[28] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Softw. Eng.*, vol. 26, no. 2, p. 20, 2021.

[29] S. Wang et al., "Automated patch correctness assessment: How far are we?" in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2020, pp. 968–980.

[30] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 226–236.

[31] M. Wen, J. Chen, R. Wu, D. Hao, and S. C. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 1–11.

[32] X.-B. D. Le, D.-H. Chu, D. Lo, C. L. Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 593–604.

[33] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Trans. Soft. Eng. Methodol. (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

[34] H. Tian et al., "Is this change the answer to that problem? Correlating descriptions of bug and code changes for evaluating patch correctness," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–13.

[35] T. Le-Cong et al., "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," 2023, *arXiv:2301.01113*.

[36] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Proc. OOPSLA*, 2007, pp. 815–816.

[37] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[38] T. B. Brown et al., "Language models are few-shot learners," 2020, *arXiv:2005.14165*.

[39] R. Li et al., "Starcoder: May the source be with you!" 2023, *arXiv:2305.06161*.

[40] Y. Liu et al., "Roberta: A robustly optimized BERT pretraining approach," 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[41] Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[42] D. Guo et al., "Graphcodebert: Pre-training code representations with data flow," 2021, *arXiv:2009.08366*.

[43] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[44] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.

[45] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing With Transformers*. Sebastopol, California: O'Reilly Media, Inc., 2022.

[46] T. L. Scao et al., "What language model to train if you have one million GPU hours?" in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL) Workshop Challenges Perspectives Creating Large Lang. Models*, 2022, pp. 765–782.

[47] B. Roziere et al., "Code llama: Open foundation models for code," 2023, *arXiv:2308.12950*.

[48] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*.

[49] C. Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[50] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.

[51] E. J. Hu et al., "Lora: Low-rank adaptation of large language models," 2022, *arXiv:2106.09685*.

[52] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," 2016, *arXiv:1508.07909*.

[53] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, Feb. 2019.

[54] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," 2021, *arXiv:2104.08821*.

[55] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur?: The manysstubs4j dataset," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 573–577.

[56] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 505–509.

[57] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *Proc. NeurIPS*, 2021, pp. 27865–27876.

[58] S. Wang et al., "Beep: Fine-grained fix localization by learning to predict buggy code elements," 2021, arXiv:2111.07739.

[59] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, 2014.

[60] "Quantization in huggingface." 2024, Accessed: Feb. 2, 2024. [Online]. Available: https://huggingface.co/docs/transformers/main_classes/quantization

[61] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. ISSTA*, 2014.

[62] V. Sanh et al., "Multitask prompted training enables zero-shot task generalization," 2022, *arXiv:2110.08207*.

[63] T. Hoang, H. J. Kang, J. Lawall, and D. Lo, "Cc2vec: Distributed representations of code changes," 2020. [Online]. Available: https://arxiv.org/abs/2003.05620

[64] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31st Int. Conf. Mach. Learn.*, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 2014, pp. 1188–1196.

[65] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus, "BEARS: An extensible java bug benchmark for automatic program repair studies," in *Proc. 26th IEEE Int. Conf. Softw. Anal., Evol. Reeng., (SANER)*, Hangzhou, China, Piscataway, NJ, USA: IEEE Press, 2019, pp. 468–478.

[66] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Trans. Softw. Eng.*, vol. 46, no. 10, pp. 1040–1067, Oct. 2018.

[67] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proc. 34th Int. Conf. Softw. Eng., (ICSE)*, Zurich, Switzerland, M. Glinz, G. C. Murphy, and M. Pezzè, Eds., Washington, DC, USA: IEEE Comput. Soc., Jun. 2012, pp. 3–13.

[68] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, Baltimore, MD, USA, M. Young and T. Xie, Eds., New York, NY, USA: ACM, 2015, pp. 24–36.

[69] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, P. Jalote, L. C. Briand, and A. van der Hoek, Eds., New York, NY, USA: ACM, 2014, pp. 254–265.

[70] J. Wei et al., "Emergent abilities of large language models," 2022, *arXiv:2206.07682*.

[71] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training LLMS on programming and natural languages," 2023, *arXiv:2305.02309*.

[72] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 201–211.

[73] Q.-N. Phung, M. Kim, and E. Lee, "Identifying incorrect patches in program repair based on meaning of source code," *IEEE Access*, vol. 10, pp. 12012–12030, 2022.

[74] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2022, pp. 654–665.

[75] J. Yang, Y. Wang, Y. Lou, M. Wen, and L. Zhang, "A large-scale empirical review of patch correctness checking approaches," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 1203–1215.

[76] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 637–661, Feb. 2020.

[77] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 959–971.

[78] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *J. Syst. Softw.*, vol. 171, Jan. 2021, Art. no. 110825.

[79] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proc. Companion ACM SIGPLAN Int. Conf. Syst., Program., Lang., Appl.: Softw. Humanity, (SPLASH)*, Vancouver, BC, Canada, G. C. Murphy, Ed., New York, NY, USA: ACM, 2017, pp. 55–56.