The Plastic Surgery Hypothesis in the Era of Large Language Models

Chunqiu Steven Xia
University of Illinois
Urbana-Champaign
chunqiu2@illinois.edu

Yifeng Ding
University of Illinois
Urbana-Champaign
yifeng6@illinois.edu

Lingming Zhang
University of Illinois
Urbana-Champaign
lingming@illinois.edu

***Abstract*—Automated Program Repair (APR) aspires to automatically generate patches for an input buggy program. Traditional APR tools typically focus on specific bug types and fixes through the use of templates, heuristics, and formal specifications. However, these techniques are limited in terms of the bug types and patch variety they can produce. As such, researchers have designed various learning-based APR tools with recent work focused on directly using Large Language Models (LLMs) for APR. While LLM-based APR tools are able to achieve state-of-the-art performance on many repair datasets, the LLMs used for direct repair are not fully aware of the project-specific information such as unique variable or method names.**

The *plastic surgery hypothesis* is a well-known insight for APR, which states that the code ingredients to fix the bug usually already exist within the same project. Traditional APR tools have largely leveraged the plastic surgery hypothesis by designing manual or heuristic-based approaches to exploit such existing code ingredients. However, as recent APR research starts focusing on LLM-based approaches, the plastic surgery hypothesis has been largely ignored. In this paper, we ask the following question: *How useful is the plastic surgery hypothesis in the era of LLMs?* Interestingly, LLM-based APR presents a unique opportunity to fully automate the plastic surgery hypothesis via fine-tuning (training on the buggy project) and prompting (directly providing valuable code ingredients as hints to the LLM). To this end, we propose FitRepair, which combines the direct usage of LLMs with two domain-specific fine-tuning strategies and one prompting strategy (via information retrieval and static analysis) for more powerful APR. While traditional APR techniques require intensive manual efforts in both generating patches based on the plastic surgery hypothesis and guaranteeing patch validity, our approach is fully automated and general. Moreover, while it is very challenging to manually design heuristics/patterns for effectively leveraging the hypothesis, due to the power of LLMs in code vectorization/understanding, even partial/imprecise project-specific information can still guide LLMs in generating correct patches! Our experiments on the widely studied Defects4j 1.2 and 2.0 datasets show that FitRepair fixes 89 and 44 bugs (substantially outperforming baseline techniques by 15 and 8), respectively, demonstrating a promising future of the plastic surgery hypothesis in the era of LLMs.

I. Introduction

The increasing complexity of source code poses a key challenge to the reliability of large-scale software systems. Software bugs in these systems can lead to safety issues [1] for users around the world as well as cause non-negligible financial losses [2]. As such, developers have to spend a large amount of effort on bug fixing. Consequently, Automated Program Repair (APR), designed to automatically generate patches to fix software bugs, has attracted wide attention from both academia and industry [3], [4], [5], [6], [7].

To achieve APR, one popular approach is known as Generate-and-Validate (G&V) [8], [9], [6], [10], [4], [11], [12], [13], [14], [15], [16], which is typically based on the following pipeline: First, fault localization techniques [17], [18], [19], [20] are applied to determine the suspicious locations in programs where bugs are likely to exist. Then, the locations are used by the APR tools to generate a list of patches that replace buggy lines with correct lines. Afterward, each patch is validated against the test suite to identify any *plausible patches* (i.e., passing all tests in the test suite). Finally, to determine the *correct patches*, developers examine the plausible patches to see if any of them can correctly fix the bug.

Traditional APR tools can mainly be categorized into heuristic-based [4], [10], [11], constraint-based [21], [22], [23], [5] and template-based [9], [12], [13], [15], [16]. Among these traditional tools, template-based APR [9], [15] have been able to achieve state-of-the-art results. Template-based APR tools typically leverage pre-defined templates (e.g., adding a nullness check) for bug fixing. However, since these fix templates are typically handcrafted, the number and types of bugs they are able to fix can be limited.

To address the limitations of traditional APR, researchers have proposed various learning-based APR tools [24], [25], [26], [27], [28], [29], [30], [31] based on the Neural Machine Translation (NMT) architecture [32] where the input is the buggy code snippets and the goal is to translate the buggy code snippets into a fixed version. To accomplish this, learning-based APR tools require supervised training datasets with pairs of both buggy and fixed code snippets in order to learn how to perform this translation step. These training data are usually obtained by mining historical bug fixes using heuristics/keywords [33], which can be imprecise for identifying bug-fixing commits; even the actual bug-fixing commits can include irrelevant code changes, leading to further pollution in the dataset [34]. Moreover, it can be hard for such APR tools to generalize and fix bug types unseen during training.

To better leverage recent advances in Large Language Models (LLMs), researchers [34], [35], [36], [37] have directly applied LLMs to generate patches without bug-fixing datasets. These LLM-based APR tools work by either infill the correct code given its surrounding context [34], [35], [38] or directly

generating a complete code function [37], [35]. By directly using LLMs that are pre-trained on billions of open-source code snippets, LLM-based APR tools have been shown to achieve state-of-the-art performance on program repair [34].

Traditional APR tools have long used the insight of the *plastic surgery hypothesis* [39] where it states that the code ingredients to fix a bug already exist within the same project. Traditional APR tools have manually designed pattern- [9], [40] or heuristic-based [41], [4] approaches to finding and using such relevant code ingredients to generate fixes for bugs. However, the plastic surgery hypothesis has been largely ignored in LLM-based APR. In fact, LLM provides a unique opportunity to fully automate the plastic surgery hypothesis idea via fine-tuning (learning project-specific information via model updates from the buggy project) and prompting (directly providing relevant code ingredients to the model), and make it directly applicable to different languages (since the LLMs are typically multi-lingual). Moreover, despite the intensive manual efforts involved, traditional APR tools still cannot fully leverage project-specific information due to large search space for leveraging/composing existing code ingredients. In contrast, the project-specific information can be effectively leveraged by LLMs due to their power in code understanding/vectorization, e.g., even partial/imprecise information may still guide LLMs in correct patch generation! To this end, we ask the question: *How useful is the plastic surgery hypothesis in the era of LLMs*?

Our Work. We present FitRepair – a LLM-based approach that automatically utilizes the plastic surgery hypothesis by systematically combining multiple fine-tuning and prompting strategies for APR. FitRepair fine-tunes LLMs using two novel domain-specific training strategies: **Knowledge-Intensified fine-tuning** – we fine-tune using the original buggy project by aggressively masking out a high percentage of tokens, which allows LLM to learn project-specific code tokens and programming styles; and **Repair-Oriented fine-tuning** – which only masks out a single continuous code sequence per training sample, allowing the model to get used to the final cloze-style APR task of predicting a single continuous code sequence. Furthermore, we directly leverage the ability for LLMs to understand natural language instructions and introduce a novel prompting strategy, **Relevant-Identifier prompting**, which uses information retrieval and static analysis to obtain a list of relevant identifiers for the buggy lines. While such relevant identifiers are critical for fixing some difficult bugs, they may not be seen by the LLM during inference due to limited context window size. Through the use of prompting, we directly tell the model to use these extracted identifiers (relevant code ingredients) to generate the correct code.

While our insight of leveraging the plastic surgery hypothesis for LLM-based APR is generalizable across different types of LLMs, to implement FitRepair, we choose a recent LLM, CodeT5 [42], which is pre-trained on millions of open-source code snippets. CodeT5 is an encoder-decoder model trained using Masked Span Prediction (MSP) objective where a percentage of tokens are masked out and each continuous masked token sequence is referred to as a masked span. To perform repair, we combine all four model variants (including the base model, both fine-tuned models and the base model with prompting). In summary, we make the following contributions:

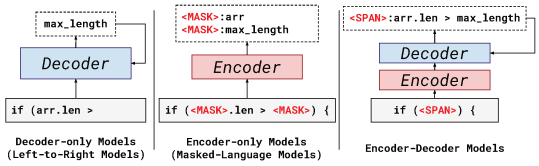

Fig. 1: LLM overview

- **Dimension.** This paper is the first to revisit the important plastic surgery hypothesis in the era of LLMs. It opens up a new dimension for LLM-based APR to incorporate previously neglected information from the buggy project itself to boost performance. Our fine-tuning and prompting strategies are crucial for boosting up LLMs for fixing the project under test, and could even enable APR for languages rarely seen or totally unseen during LLM pretraining.
- **Implementation.** We implement FitRepair based on the recent CodeT5 model. We augment the model using two novel fine-tuning strategies: Knowledge-Intensified fine-tuning and Repair-Oriented fine-tuning, along with a novel prompting strategy based on information retrieval and static analysis: Relevant-Identifier prompting. We combine the patches generated by all four models together and perform patch ranking to speed up APR.
- **Evaluation Study.** We conduct an extensive evaluation against state-of-the-art APR tools. On the widely studied Defects4j 1.2 and 2.0 datasets [43], FitRepair is able to achieve the new state-of-the-art results of 89 and 44 correct bug fixes (15 and 8 more than best baseline) respectively. Furthermore, we perform a broad ablation study to justify our design. FitRepair demonstrates for the first time that the plastic surgery hypothesis can substantially boost LLM-based APR, while being fully automated and general. Moreover, even partial/imprecise code ingredients may still effectively guide LLMs for APR!

II. Background

A. Large Language Model

Large Language Models (LLMs) trained on large amounts of text combined with code snippets from a broad range of open-source repositories have shown impressive progress in various code-related tasks [44], [45], [46]. On top of that, LLMs can be *fine-tuned* (updatingLLM parameters by further training on the downstream tasks [47]) to target a specific task such as software testing [48], defect prediction [49] or code clone detection [50]. Different from fine-tuning, *prompting* is a way to directly use LLMs on downstream tasks without further training. Prompting involves providing a natural language description of the task and optionally including a few demonstrations of the task to the LLMs before the actual input. Researchers have successfully applied prompting to tasks like code completion [51] and code summarization [52].

LLMs are based on the popular Transformer architecture [53], which combines an **encoder** with a **decoder** to perform text generation. The encoder first takes in the input to the model and then produces an encoded representation. The decoder uses this encoded vector to autoregressively generate the next token based on all previously generated tokens. Using this paradigm, researchers build larger and larger models (as large as 540B in the number of model parameters [54]) and demonstrated impressive results on code-related tasks, especially on program synthesis [44], [55], [56], [57], [58].

LLMs can be classified into three groups based on their model architecture and pre-training objective: **Decoder-only** (Left-to-Right Language Models), **Encoder-only** (Masked Language Models), and **Encoder-decoder models**. Figure 1 shows an overview of the three different LLM architectures. Decoder-only models perform left-to-right generation by producing the probability of a token given all previous tokens. One of the most well-known LLMs, GPT [59], [60], is based on this architecture. During training, decoder-only models aim to predict the next token given all previous context. Examples of decoder-only models for code are CodeGPT [61], CodeParrot [62], and Codex [44]. These models can be directly used for program generation given previous code contexts. Encoder-only models, on the other hand, only use the encoder component to provide an encoded representation of the input. Models such as BERT [47] are trained using the Masked Language Model (MLM) objective, where a small percentage (e.g., 15%) of the training tokens are masked out and the model aims to recover these masked tokens using the bi-directional context. CodeBERT [63], GraphCodeBERT [64], and CuBERT [65] are examples of encoder-only models where it can provide a representation of the input code to be used for downstream tasks such as code clone detection [50]. Encoder-decoder models (T5 [66], BART [67]) use both components of the transformer and are typically trained using Masked Span Prediction (MSP) objective. Different from MLM, instead of masking out individual tokens, MSP replaces a sequence of tokens with a single span mask. The goal of the training is to recover the original sequence using both the context before and after the span mask token. CodeT5 [42] and PLBART [68] are examples of encoder-decoder models and due to the MSP pre-training objective, they can be directly used to fill in arbitrary code snippets given the bi-directional code context.

B. Automated Program Repair

Automated Program Repair (APR) works by automatically generating patches when given the buggy project and potential fault locations. Traditional APR tools can be categorized into constraint-based [21], [22], [23], [5], heuristic-based [4], [10], [11], and template-based [9], [12], [13], [15], [16] tools. Among those, template-based APR has been regarded as the state-of-the-art in achieving the best repair performance [9], [15]. Template-based APR works by using pre-defined templates (handcrafted by human experts) which target specific patterns in source code. Each template will have an associated fix that modifies the found patterns in the source code to fix specific types of bugs. However, template-based APR tools cannot fix bugs that do not fall under the pre-defined templates. As a result, template-based tools lack the ability to generalize to unseen bug types.

In recent years, researchers have begun to focus on learning-based APR approaches such as TENURE [31], Tare [30], SelfAPR [69], RewardRepair [29], Recoder [28], CURE [26], and CoCoNuT [27] based on the Neural Machine Translation (NMT) [32] architecture. The goal of these tools is to learn a transformation using DL models that turns buggy code snippets into patched ones. To facilitate this, these tools require further training on specific bug-fixing datasets containing pairs of buggy and fixed code snippets. However, as discussed in prior work [34], these bug-fixing datasets are usually scraped from open-source bug-fixing commits using handwritten heuristics such as keyword searching [28], [27], [33], [70], which can include irrelevant code commits; even the correctly identified bug-fixing commits may contain various irrelevant code changes (such as refactoring or new feature implementation), introducing various noises in the datasets. Also, to avoid including bug-fixing commits with irrelevant code changes, existing learning-based APR techniques will limit the commits to ones with few lines of changes [26], [28], [27], further limiting the amount of training data. Moreover, NMT-based APR may still not generalize to specific code or bug types unseen inside of the (limited) bug-fixing datasets.

Recognizing these limitations in NMT-based APR, researchers have proposed LLM-based APR tools which do not require bug-fixing datasets by directly using LLMs for APR. AlphaRepair [34] reformulated the APR problem as a cloze (or infilling) task to directly leverage LLMs in a zero-shot manner to fill in the code given the context before and after the buggy line, and demonstrated that LLMs can directly outperform all prior APR techniques. Other studies [35], [36], [37], [38] also used different LLMs (including Decoder-only and Encoder-decoder models) to not only perform cloze-style APR but also repair scenarios where a complete fixed function is generated. Contrasting with NMT-based APR tools, LLM-based APR leverages the pre-training objectives of LLMs which can directly learn the relationship between correct code and its context without relying on historical bug-fixing commits. As a result, LLM-based APR tools have shown to achieve state-of-the-art performance on repair tasks across multiple programming languages [35]. Even more recently, there has been concurrent work on leveraging the popular dialogue-based LLMs (e.g., ChatGPT [71]), which are fine-tuned using reinforcement learning from human feedback (RLHF) [72], to build advanced APR tools [73], [74], [75]. Such APR tools can directly take in test execution feedback to not only learn from the immediate feedback/hint (i.e., single-turn) [74] but also leverage the full conversational history to make use of multiple previous feedback (i.e., multi-turn) to generate a new patch [73], [75]. Orthogonally, in this work, we present the first work to further advance state-of-the-art LLM-based APR with the insight of the plastic surgery hypothesis.

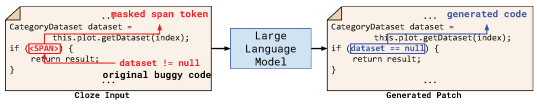

Fig. 2: Cloze-style APR

III. Approach

In this section, we describe our approach to incorporate the plastic surgery hypothesis for LLM-based APR. While our overall idea is general for LLM-based APR approaches, we mainly focus on cloze-style APR [34] since it has been demonstrated to be the state-of-the-art for LLM-based APR [35]. Figure 2 shows the overview of cloze-style APR where we aim to directly generate the correct code in place given the original context, where the model has to "fill in the blanks" of the missing code line/hunk (represented by `<SPAN>`) given the buggy context. However, prior cloze-style APR tools largely ignore the rich project- and bug-specific information that has been demonstrated by the plastic surgery hypothesis [39] to be critical in helping to fix the bug. LLM presents an opportunity to automatically leverage the idea of the plastic surgery hypothesis via its powerful ability to both directly learn from the buggy project (fine-tuning) and use relevant code context clues (prompting) to generate correct fixes. In this work, we explore using LLMs to automatically capture project-specific information via both fine-tuning and prompting.

We propose a novel approach – FitRepair to combine the direct usage of LLM in cloze-style APR with knowledge gained from the plastic surgery hypothesis. FitRepair first trains two separate models with two novel fine-tuning strategies: **1) Knowledge-Intensified fine-tuning** – we use the source files of the original buggy project to construct a similar dataset to pre-training by aggressively masking out large portions (50%) of the code tokens, which allows LLM to learn project-specific code tokens and programming styles; and **2) Repair-Oriented fine-tuning** – we fine-tune another model using the original buggy project to construct a more repair-oriented dataset by masking out only a single continuous code sequence per training sample, which makes the fine-tuned model become more prepared for the repair task where only a single continuous code sequence needs to be generated. Additionally, we propose a novel prompting strategy, **3) Relevant-Identifier prompting**, by obtaining a list of relevant/rare identifiers that are not seen by the model in its immediate context using information retrieval and static analysis.

While our approach can be extended to different LLMs, in this paper, we use CodeT5 [42], an encoder-decoder LLM for code trained using Masked Span Prediction (MSP) objective. MSP replaces continuous tokens with a single masked span token (`<SPAN>`) and the pre-training task is to recover masked-out code sequences given the surrounding context. Given a sequence of tokens $X = \{x_1, ..., x_n\}$, random sequences of tokens are replaced with a masked span token to produce $X_{masked} = \{x_1, ..., \texttt{<SPAN>}, x_n\}$. Let $M = \{m_1, ..., m_k\}$ be the tokens masked out, $M_{<g} = \{m_1, m_2, ..., m_{g-1}\}$ be token sequence predicted so far where $g \leq k$, P be the predictor (model) which outputs the probability of a token. The MSP loss function can be described as: $\mathcal{L}_{MSP} = -\frac{1}{k}\sum_{i=1}^{k} log\ (P\ (m_i \mid X_{masked},\ M_{<i})$

We follow previous work [34] and use repair templates to generate mask lines where we replace the entire or parts of the buggy line with a single masked span token. We then use CodeT5 to generate the correct code to replace the masked span and create a patch for the bug. Figure 3 shows the overview of our approach:

- ❶ **(Section III-A)**: We use Knowledge-Intensified fine-tuning to build a training dataset by extracting functions from the buggy project. We fine-tune the original CodeT5 model by first using a high masking rate to aggressively learn project-specific tokens.
- ❷ **(Section III-B)**: We use Repair-Oriented fine-tuning strategy to fine-tune another model by constructing another training dataset from the buggy project where only a single or partial code line is masked out based on the repair templates. We train the model until convergence and obtain the Repair-Oriented fine-tuning model.
- ❸ **(Section III-C)**: We use Relevant-Identifier prompting strategy to extract relevant identifiers via information retrieval and static analysis. We then create individual prompts with instructions for the model to use the extracted identifiers during patch generation.
- ❹ **(Section III-D)**: We perform cloze-style APR by using the repair templates from previous work [34] and generate patches by separately using the 4 models (original CodeT5, Knowledge-Intensified fine-tuning model, Repair-Oriented fine-tuning model, and original CodeT5 with prompting). Each patch generated is then validated against the test suite to find the list of plausible patches.

A. Knowledge-Intensified Fine-tuning

To facilitate the learning of project-specific information, we use Knowledge-Intensified fine-tuning by constructing a training dataset using the buggy project itself. Figure 4 shows the Knowledge-Intensified fine-tuning process. We first extract the source code functions from the project code base where the bug is from and apply MSP objective – masking out multiple spans of code tokens, used to pre-train the original CodeT5 model. Traditionally, MSP objective will mask out only a small portion of the original code tokens (e.g., 15% [42]). However, in this step, we employ a much higher masking rate (50%) which means the model is tasked with recovering more masked-out code tokens with less context. This approach is motivated by a recent study [76] on LLMs for natural languages where better representation and downstream performance can be achieved by increasing the pre-training masking rate of MLM and MSP objectives. The study found that higher masking rates make the learning tasks more challenging and can force the model to learn more aggressively, which helps improve the performance of LLMs on various downstream tasks. In this paper, we leverage this idea of more aggressive training to force the model to learn more project-specific knowledge by trying to recover more code tokens given limited context.

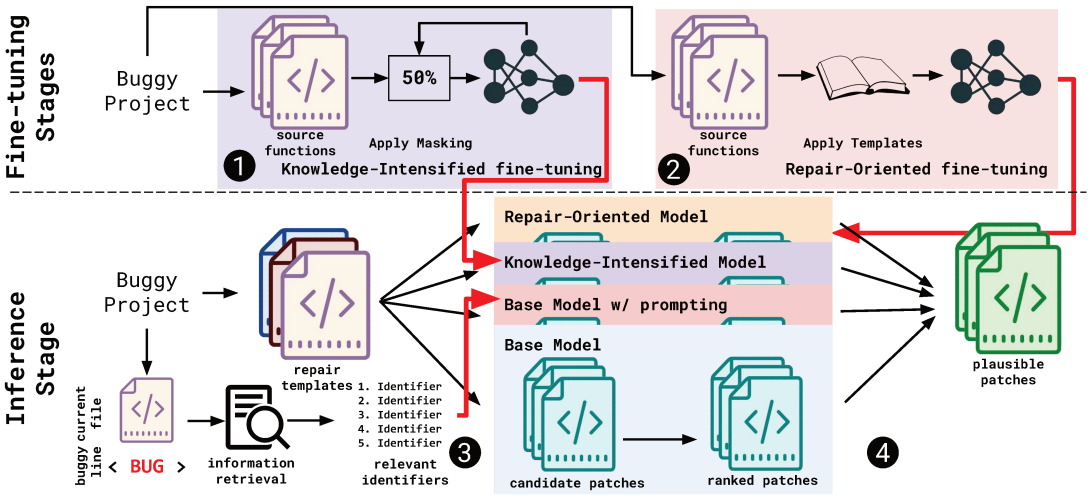

Fig. 3: FitRepair overview

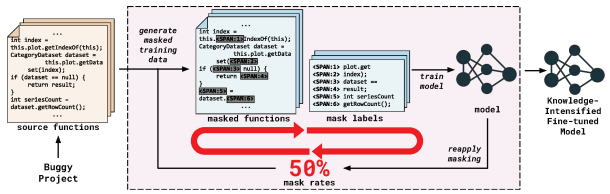

Fig. 4: Knowledge-Intensified fine-tuning overview

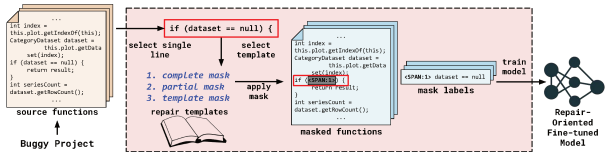

Fig. 5: Repair-Oriented fine-tuning overview

However, one limitation with fine-tuning the model on the buggy project itself is the relatively small number of training samples (e.g., thousands of functions) especially compared with the large amount of open-source pre-training data used in CodeT5 (millions of functions). As a result, we *reapply* the MSP objective across iterations to augment the fine-tuning dataset with more training samples. Following the example in Figure 4, we start by creating one set of training data by masking out 50% of the training tokens to create masked spans. In the next iteration, we reapply this masking strategy to create a new set of training data by randomly choosing another 50% to mask out again. In this process, we essentially create new training data for each subsequent iteration. While the number of tokens masked out is the same, the specific masked locations can be different which provides further augmentation on the training dataset allowing the model to learn more project-specific tokens.

During the fine-tuning process when using Knowledge-Intensified fine-tuning, FitRepair is able to learn project-specific knowledge such as commonly used methods or variables that are specifically defined in the current project. These pieces of project-specific knowledge are especially important for repair as many bugs can be fixed by applying code snippets found in other parts of the source file or project, according to the plastic surgery hypothesis [39]. Due to the limited context window size (e.g., 512 tokens for CodeT5), CodeT5 cannot encode all of the surrounding contexts during inferencing, which leads to the base CodeT5 model missing variable names and method calls used in other parts of the context that are actually necessary to be used as part of the patch. Knowledge-Intensified fine-tuning can partially alleviate this by learning these missing variable names and method calls as part of the fine-tuning such that when used for cloze-style APR, the fine-tuned model can predict these useful tokens with a higher probability compared to the model without fine-tuning. While Knowledge-Intensified fine-tuning can help with better learning of project-specific details, we next introduce Repair-Oriented fine-tuning which produces another model that aims to optimize for the repair task.

B. Repair-Oriented Fine-tuning

In the previous step, we use Knowledge-Intensified fine-tuning to generate more code that uses project-specific variables, method calls, and structures. Cloze-style APR frames the problem of patch generation by asking the LLM to fill in the correct code given the buggy context. This is achieved by exploiting the similarity between the pre-training objective and final inference setup to generate patches. Furthermore, cloze-style APR is not limited to only generating a complete line but additionally using repair templates (e.g. replaces only method call name) which keeps part of the buggy line to generate partial lines [34]. However, both the Knowledge-Intensified fine-tuning CodeT5 and the original CodeT5 suffer from the same limitation: *the training process is not designed for repair*. Both the original pre-training and Knowledge-Intensified fine-tuning use MSP which masks out multiple disjointed code token spans. The goal of the model during training is to recover the original tokens for all the masked spans. However, for cloze-style APR, only a single code line or a part of the code line is usually masked out and the model only has to predict the correct code for that single span.

To address such limitations, we use Repair-Oriented fine-tuning which fine-tunes with a training setup that is similar to

the repair inference task. Figure 5 shows the detailed Repair-Oriented fine-tuning process. We again first use the original buggy project as the source of our training data by extracting source code functions. We pick a single code line in each training sample to mask out with a single span token. One can think of this chosen code line as the buggy line in the final repair scenario. To model the impact of eventually using template-based repair inputs, we randomly select a repair template that can be applied on this line to mask out only a part of the line. These repair templates are taken directly from previous cloze-style APR work [34] and can be categorized into 3 different types of repair templates: **1) Complete mask** – replace the entire buggy line with a single span token or add the span token to before/after the buggy line, **2) Partial mask** – keep some original buggy line tokens at the end or beginning of the line and replace the rest with a span token, and **3) Template mask** – target specific code line types by replacing the method call, method parameters, and Boolean expression or operator with a span token. In this fine-tuning process, our training samples are closely similar in their setup compared with the cloze-style APR task that we want the model to perform. Using Repair-Oriented fine-tuning, we can produce a fine-tuned model which aims to follow closely with the final repair task.

C. Relevant-Identifier Prompting

Two previous fine-tuning strategies aim to fine-tune the model towards generating more project-specific tokens and get used to repair-oriented inputs by using the original buggy project as the fine-tuning dataset. However, this means that the two fine-tuned models have been geared toward the entire buggy project rather than the specific bug within the project. For specific bugs, the relevant code ingredients may be drastically different depending on the bug file, location, and type of buggy line. These ingredients can be possibly far away from buggy locations, making it hard for them to be included in the input context due to the **limited context window size** of LLMs (e.g. the context window size of CodeT5 is 512 tokens).

We use Relevant-Identifier prompting on the base CodeT5 model to directly prompt the model with relevant code ingredients. Algorithm 1 illustrates our prompting strategy. Given the buggy line information, we first extract the file containing the bug and separate it into individual code lines (Line 1). Prior work has found that a significant percentage of the correct code to fix the bug can be found within the same file [39]. We then use Levenshtein Distance Ratio [77] (other string comparison methods [78], [79], [80] provide similar results) to measure the similarity between each line compared with the buggy line (Line 4). The hypothesis is that useful identifiers can be obtained from lines that are very similar to the buggy line [81]. We rank each code line based on its string similarity score from high to low to get a ranked list of code lines (Line 5). Since we want to provide the model with identifiers to help generate the correct fix, we extract identifiers from each line (Line 7). We perform further filtering by first removing any common/simple identifiers (e.g., `length` and `node`) (Line 8)

Algorithm 1 Relevant-Identifier Prompting Strategy

Inputs: Buggy project, file and line: $Proj$, $File$, $buggy\_line$
Output: Relevant-Identifier: $prompts$

1: $lines$:= ExtractLines($File$)
2: $similarities$, $identifiers$:= [ ], [ ]
3: **for** $line$ in $lines$ **do**
4: $similarities$.append(LevenshteinRatio($buggy\_line$, $line$))
5: $lines\_ranked$:= RankLines($lines$, $similarities$)
6: **for** $line$ in $lines\_ranked$ **do**
7: $line\_identifiers$:= ExtractIds($line$)
8: $identifiers$.extend(SimpleFilter($line\_identifiers$))
9: $accessibles$:= FindAccessIds($Proj$, $File$, $buggy\_line$)
10: $relevants := identifiers \cap accessibles$
11: $type\_infos$:= FindTypeInfo($Proj$, $relevants$)
12: $prompts$:= BuildPrompts($relevants$, $type\_infos$)

and then using static analysis (Line 9) to remove any identifiers that are inaccessible within the buggy method (Line 10). Next, we extract the useful type information for each identifier and if it is a method invocation or a variable (Line 11). Finally, we obtain a ranked list of complex identifiers that come from similar lines within the same file.

We then generate the prompts to instruct the model to use these extracted identifiers to generate patches (Line 12). LLMs are able to understand natural language instructions to perform specific tasks. In Relevant-Identifier prompting, we construct a prompt in the form of `/* use {} in the next line */` where we replace {} with an identifier with type information (e.g., `(Plot) getParent()`). This prompt is then appended before the masked span token during inference that allows the model to directly use this identifier information provided in its generation. Since we have a ranked list of identifiers, we generate multiple unique prompts, each including one of the highest-ranked identifiers. By directly providing these extracted bug-specific identifiers in prompts, the model can use these identifiers which previously are outside of its immediate context to generate the correct fix.

D. Patch Generation, Ranking, and Validation

We directly use the base CodeT5 model (with and without prompting) and the two fine-tuned models generated from previous steps for patch generation. To generate patches to replace the buggy lines, we apply repair template inputs from previous work [34] and ask each model to fill in the masked-out span token with generated correct code. Following prior work [37], [35], [36], we sample each model in parallel to generate its own set of patches. In total, for each bug, we generate four lists of potential patches using the four models.

The rankings of patches are computed based on the outputs of each model. We follow the same process as previous work [34] and compute the likelihood score. For each candidate patch, we want to provide a likelihood score that can accurately reveal the extent to which CodeT5 will generate this patch. Let $T = \{t_1, t_2, ..., t_n\}$ be the list of tokens generated for a patch and $C(t_i)$ be the probability of generating token t_i according to CodeT5, then the likelihood score is defined as: $score(T) = \frac{1}{n} \sum_{i=1}^{n} \log\left(C(t_i)\right)$.

We compute this likelihood score for all the patches generated across all the templates and re-rank patches from the highest score to the lowest score. We then validate each candidate patch in accordance with the ranking results. Since each model (two fine-tuned models, base CodeT5 with and without prompting) generates its own separate list of patches, we use a round-robin schedule to validate one patch from each model before rotating. Note, we can also perform the patch validation in parallel. As such, we can reduce the patch validation time by stopping after any one of the models found a correct patch according to manual inspection by developers.

IV. Experimental Design

A. Research Questions

In this paper, we study the following research questions:

- **RQ1**: How does FitRepair compare against the state-of-the-art APR tools?
- **RQ2**: What is the impact of different configurations of FitRepair?
- **RQ3**: How does FitRepair generalize in fixing additional bugs from different projects?

We first demonstrate the repair effectiveness of FitRepair against state-of-the-art APR tools on the popular Defects4j 1.2 [43] dataset. We study not only the number of bugs fixed in total but also the number of unique bugs fixed compared with previous techniques. Furthermore, we analyze the improvement in patch ranking – to validate correct patches faster when using FitRepair. Next, we conduct an extensive ablation study on the different configurations of both our two fine-tuning strategies and one prompting strategy. Due to the time cost to train multiple models, for the ablation study, we focus on the `Closure` project in Defects4j which is the largest in both the number of bugs and source code size. Following prior work [34], [35], we evaluate against the state-of-the-art APR tools on the Defects4j 2.0 [43] dataset to illustrate that FitRepair is not simply overfitting to the 1.2 version.

B. Implementation

FitRepair is implemented in Python using the PyTorch [82] implementation of the CodeT5 model from Hugging Face [83]. Our fine-tuning method is based on the pre-trained CodeT5-large (770M) checkpoint. We use JavaParser [84] to perform static analysis of filtering inaccessible identifiers in scope. For both Knowledge-Intensified fine-tuning and Repair-Oriented fine-tuning, we repeat the fine-tuning process for 10 iterations by default to augment our fine-tuning dataset. We fine-tune the CodeT5 model on an NVIDIA RTX A6000 with 48GB memory using FP16. We use the following set of hyper-parameters to train models: 32 batch size and 1e-4 learning rate with 15K training steps. We use Adam optimizer [85] to update the parameters and use a linear learning rate scheduler with 10% warmup proportion. For both fine-tuning strategies, we extract the oldest version of the buggy project for training and use the fine-tuned models to generate patches for all bugs in that project. For Relevant-Identifier prompting, we use the top 5 most relevant identifiers. For repair, we sample each model 5000 times and validate the top 1000 unique patches produced by each model (at most 4000 patches in total per bug) which is comparable to other baselines. To generate more unique patches for each sample, we use nucleus sampling with top p of 1 and temperature of 1. We validate the patches on a workstation using AMD Ryzen Threadripper PRO 3975WX CPU with 32-Cores and 256 GB RAM, running Ubuntu 20.04.5 LTS. Similar to prior work [34], [26], [24], we use an end-to-end time limit of 5 hours to fix one bug. Note that we sum up the time spent on each of our four processes as FitRepair time cost for fair comparison.

C. Subject Systems

We use the widely studied benchmark of Defects4j [43] – a collection of open-source bugs found across 15 different projects to evaluate FitRepair. We follow prior work and separate the dataset into Defects4j 1.2 and 2.0. Defects4j 1.2 contains 391 bugs across 6 different projects and Defects4j 2.0 contains 438 bugs across 9 additional projects. While evaluating FitRepair on all 391 bugs in Defects4j 1.2, we follow prior work [34] and choose only the 82 single-line bugs in Defects4j 2.0 for evaluation (since existing learning-based APR mainly target single-line fixes).

D. Compared Techniques

We compare FitRepair against 20 different APR tools including both state-of-the-art learning-based and traditional APR tools. We choose 8 recent learning-based APR tools: AlphaRepair [34], SelfAPR [69], RewardRepair [29], Recoder [28], CURE [26], CoCoNuT [27], DLFix [24] and SequenceR [25]. AlphaRepair is a recently proposed and state-of-the-art cloze-style APR tool that directly uses an LLM (CodeBERT). Additionally, we also compare against 12 representative traditional APR tools: TBar [15], PraPR [9], AVATAR [16], SimFix [41], FixMiner [14], CapGen [11], JAID [86], SketchFix [12], NOPOL [23], jGenProg [87], jMutRepair [13], and jKali [13]. Finally, since FitRepair proposes to combine the base CodeT5 (with and without prompting) with the two fine-tuned models, we also compare against a baseline where we run the base CodeT5 four times with four random different seeds. This is a fair and necessary baseline to compare against as CodeT5 can produce different sampling outputs depending on the random seed and a developer who wishes to use our approach of combining the four models together may also allocate the same GPU resource to run CodeT5 four times as well. We refer to this baseline in our evaluation as CodeT5$\times$4.

We evaluate against these baselines on *perfect fault localization* setting, where the ground-truth location of each bug is provided to the repair tool by comparing the reference developer patch with the buggy code. This is the preferred evaluation setting [27], [26], [28], [7] as it eliminates any result differences caused by using different fault localization techniques [17]. We use the standard metrics for APR comparison of *plausible patches* – pass the entire test suite and *correct patches* – semantically equivalent to the reference

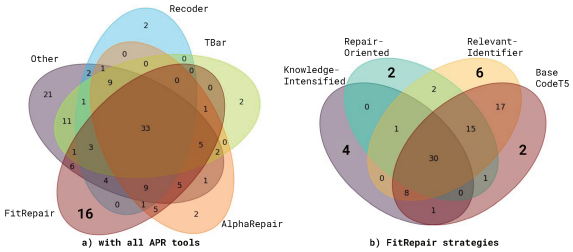

Fig. 6: Correct fix Venn diagram on Defects4j 1.2

```
  if (objectType != null) {
-   boolean isOverride = t.inGlobalScope() &&
    /* use (Node) getJSDocInfo() in the next line */
+   boolean isOverride = parent.getJSDocInfo() != null &&
        parent.getType() == Token.ASSIGN &&
```

Bug: *Closure-71* a)

```
  MockHandlerInterface<T> oldMockHandler = getMockHandler(mock);
- MethodInterceptorFilter newFilter = new MethodInterceptor...
   /* use (MockUtil) getMockHandler() in the next line */
+  MockHandler<T> newMockHandler = new MockHandler<T>(oldMockHandler);
+  MethodInterceptorFilter newFilter = newMethodInterceptorFilter(
+      newMockHandler.getMockSettings());
   ((Factory) mock).setCallback(0, newFilter);
```

Bug: *Mockito-27* b)

Fig. 7: Relevant-Identifier prompting unique patches

developer fix. Following all prior APR work, correct patches are determined by manually inspecting each plausible patch. Also, following common practice in APR, we directly report the number of correct and plausible bug fix results from previous studies [34], [9], [15], [28], [29].

V. Result Analysis

A. RQ1: Comparison with State-of-the-art

1) Bugs fixed: We first compare FitRepair against both traditional and learning-based APR tools on Defects4j 1.2. Table I shows the number of bugs that can be fixed with correct patches by FitRepair and the top baseline tools. In addition to FitRepair, we also include the result of running the base CodeT5 separately four times using four different seeds (Column CodeT5×4). Compared with CodeT5×4, we observe that our fine-tuned models and prompting strategy is able to provide additional fixes, boosting the number of correct bug fixes from 80 to 89. In total, *FitRepair is able to achieve 89 correct bug fixes on Defects4j 1.2 with 15 more fixes than the current state-of-the-art APR tool.* Figure 6a shows the number of unique bug fixes (that only one technique can exclusively fix while others cannot) generated by FitRepair compared with the top performing APR baselines and all other tools (Other). We observe that even compared with all previous APR approaches, FitRepair is able to *provide 16 additional unique fixes that no other APR tools have been able to fix so far on Defects4j 1.2*.

To illustrate the ability of FitRepair, we show an example fix on a bug (`Closure-71`) in Figure 7a which cannot be fixed by any previous tools. The fix is to invoke the method of `getJSDocInfo()` which is not only an uncommon method name but also is not used in the surrounding context. However, this method has been used within the same file as the buggy code where another function initializes a variable called `overridingInfo` also using `getJSDocInfo()`. Relevant-Identifier prompting is able to recognize the similarity between the buggy variable name (`isOverride`) and this line to extract

```
      int p = NodeUtil.precedence(type);
-     Context rhsContext = Context.OTHER;
+     Context rhsContext = getContextForNoInOperator(context);
      addExpr(first, p + 1, context);
```

Bug: *Closure-123*

Fig. 8: Knowledge-Intensified fine-tuning unique patch

the relevant identifier of `getJSDocInfo()` and provide the prompt to tell the model to directly use this identifier to generate the correct patch. This example showcases the plastic surgery hypothesis where patches can often be constructed via reusing code snippets/ingredients from other parts of the project. FitRepair directly leverages this hypothesis by extracting the relevant identifiers from similar code lines within the current file and providing it via natural language prompting to generate the correct fix.

Another interesting example bug (`Mockito-27`) fixed by FitRepair that cannot be fixed by previous tools is in Figure 7b. This bug is fixed by calling the `MethodInterceptorFilter` constructor with a previous setting obtained using `getMockSetting()`. We observe that while the Relevant-Identifier prompting did not get the exact identifier of `getMockSetting`, it was able to gather a closely related identifier of `getMockHandler`. Due to their power in code understanding, LLMs do not have to always generate a patch containing the exact identifier in the prompt and can often just use it as hints or partial code for generation. This example further highlights the unique effect the plastic surgery hypothesis have on LLM-based APR where the extracted code ingredients do not have to be exactly correct and can serve as guidance for the model to generate the correct patch.

2) Individual strategy effect: Figure 6b shows the unique bugs fixed by each of the individual strategies in FitRepair. We observe that our three strategies are all able to contribute to providing unique fixes compared with the base CodeT5 model (4 from Knowledge-Intensified fine-tuning, 2 from Repair-Oriented fine-tuning and 6 from Relevant-Identifier prompting) and boost the overall FitRepair to achieve 89 correct patches. Interestingly, each single strategy is already a strong APR tool, e.g., the Relevant-Identifier prompting strategy can fix 79 bugs by itself, already outperforming all existing tools. This demonstrates the ability of our three strategies to provide additional fixes that directly applying the base CodeT5 cannot provide. Moreover, the base CodeT5 model without any changes also produced 2 unique fixes, demonstrating the usefulness of applying the base model to cover the bugs that may only require general-purpose correct code knowledge.

Since Section V-A1 has shown unique fixes obtained by Relevant-Identifier prompting, we now present an example bug fixed by Repair-Oriented fine-tuning. Figure 8 shows a correct fix example that base CodeT5 model cannot fix on `Closure-123`. For this bug, the correct fix is to initialize the variable `rhsContext` by calling a function `getContextForNoInOperator()`. What makes this bug difficult to fix for previous APR tools is that `getContextForNoInOperator` is a very hard sequence to generate. First, it is not a commonly used function name such as `getContext`. Second, there are no code snippets using

TABLE I: Evaluation results of correct fixes on Defects4j 1.2

Project	FitRepair	CodeT5×4	AlphaRepair	SelfAPR	RewardRepair	Recoder	TBar	CURE	CoCoNuT	PraPR	DLFix
Chart	8	8	9	7	5	10	11	10	7	7	5
Closure	29	23	23	19	15	21	16	14	9	12	11
Lang	19	18	13	10	7	11	13	9	7	6	8
Math	24	23	21	22	19	18	22	19	16	10	13
Mockito	6	5	5	3	3	2	3	4	4	3	1
Time	3	3	3	3	1	3	3	1	1	3	2
Total	**89**	80	74	64	50	65	68	57	44	41	40

TABLE II: Average correct patch rank on Defects4j 1.2

Project	Chart	Closure	Lang	Math	Mockito	Time	Average
CodeT5×4	34	510	803	693	1058	787	618
FitRepair	20	414	356	448	378	271	363
Improvement	41%	19%	56%	35%	64%	66%	41%

this function in the immediate context. As such, previous techniques may fail to generate this patch as it requires specific knowledge about the buggy project in order to come up with this function name. However, this function is used multiple times within the buggy project (in other functions and files). FitRepair leverages this by using Knowledge-Intensified fine-tuning strategy to fine-tune a model to predict masked-out tokens within the buggy project. During Knowledge-Intensified fine-tuning, the model can learn the usage of this specific function within the buggy project and apply it in this case to produce the correct patch. This domain-specific knowledge cannot be learned just from pre-training on a large amount of open-source code (previous cloze-style APR tools).

3) Patch ranking: We examine the ability of FitRepair to perform patch ranking in order to prioritize faster validation for correct patches. Similarly, we compare FitRepair against the baseline of running base CodeT5 four times with different seeds. Table II shows the average rank of the correct patch for the Defects4j 1.2 projects on the same set of bugs both FitRepair and CodeT5×4 can fix. We observe that in all six projects, FitRepair provides a better rank on average for the correct patches. On average, using FitRepair, we can achieve a 41% reduction in the ranking of correct patches. FitRepair can learn/use project-specific information to rank correct patches higher since the correct patches often use project-specific identifiers which are less prioritized by the base CodeT5 model. In this way, FitRepair not only fixes more bugs, but can also find the correct fixes faster and reduce the computation cost needed for patch validation.

B. RQ2: Detailed Ablation Study

TABLE III: Repetition for Knowledge-Intensified

Strategy	#Corr. / #Plaus. (All)	#Corr./#Plaus. (New)	Comp. Error pct	#Unique comp. per bug
Repetitive (default)	**15 / 30**	**2 / 3**	82%	**138**
Non-Repetitive	14 / 25	1 / 2	**79%**	104

TABLE IV: Masking rates of Knowledge-Intensified

Mask Rate	#Corr. / #Plaus. (All)	#Corr./#Plaus. (New)	Comp. Error pct	#Unique comp. per bug
10%	**16** / 23	**2** / 2	83%	79
20%	14 / 27	**2** / 4	88%	75
30%	14 / 30	**2** / 4	87%	92
40%	**15** / 29	**2** / 4	85%	102
50% (default)	**15** / 30	**2** / 3	**82%**	**138**
60%	13 / 26	1 / 4	87%	106
70%	12 / 21	1 / 2	88%	81
80%	9 / 17	**2** / 2	86%	88
90%	7 / 13	**2** / 3	93%	47

1) Impacts of Knowledge-Intensified fine-tuning: The goal of training using Knowledge-Intensified fine-tuning is to incorporate more project-specific knowledge to CodeT5. There are two hyper-parameters of Knowledge-Intensified fine-tuning, including **mask rate** and **repetition iterations**. Our default setting is to use a 50% mask rate and 10 repetition times. We conduct an ablation experiment to study the impacts of these two hyper-parameters on the number of correct/plausible patches generated, the number of unique bugs fixed (via correct/plausible patches) when compared to the base CodeT5 model, the compilation error rate, and the number of unique compilable patches generated per bug.

We first examine the impact of repeating the masking multiple times during training to generate additional training samples. Table III shows the results on the Closure bugs with 10 repetition iterations – generating 10 training sets (Row Repetitive) and no repetition iterations – generating only 1 training set (Row Non-Repetitive). We observe that the number of total correct and plausible patches produced by the repetitive training approach is higher. Additionally, when we generate new masked training samples during each iteration, the model produced is able to generate two unique bug fixes compared with the base CodeT5 model. Similar results can be found when we look at the compilation error rate together with the number of unique compilable patches generated. We see that while the non-repetitive approach has a lower compilation error rate, the number of compilable patches generated is much less. By repeating the masking multiple times during training, we are able to fine-tune the model to learn more project-specific information to produce compilable patches and to fix more unique bugs.

Next, we study the impacts of different mask rates. In this experiment, we use the default of 10 repetition iterations by generating 10 unique training samples during fine-tuning and examine how different mask rates can have on performance. Due to the extremely large search space (considering unlimited choices of mask rates), we choose mask rates from 10% to 90% with an interval of 10%. Table IV shows our experimental results on the bugs in the `Closure` project. First, we observe that an extremely high mask rate (70, 80, 90%) performs poorly in terms of the number of bugs fixed and compilation rate. While a high mask rate may force the model to learn more project-specific tokens during training, each training sample will have a majority of its tokens masked out. Compared with the final repair task of generating a single or partial line, the extremely high mask rate makes the resulting model ill-suited for repair. We observe that the default setting of 50% mask rate strikes a good balance between achieving the high total

number of bugs fixed, more unique bugs fixed compared to base CodeT5, and a relatively low compilation error rate. By using a balanced mask rate of 50%, the Knowledge-Intensified fine-tuning model is able to best complement the base CodeT5 in generating more unique bug fixes.

TABLE V: Masking strategies of Repair-Oriented

Strategy	AST masking	Single-line masking	Template masking
#Corr. / #Plaus. (All)	**13** / 25	10 / 21	**12** / 23
#Corr. / #Plaus. (New)	0 / 1	**1** / 1	**1** / 2
Comp. Error pct	88%	86%	**76%**
#Unique comp. per bug	101	129	**139**

2) Impacts of Repair-Oriented fine-tuning: In addition to looking at the impact of different configurations when using Knowledge-Intensified fine-tuning, we study the different ways we can apply Repair-Oriented fine-tuning. Specifically, we design two additional strategies that can be used during training to produce masked training samples. Table V shows the result of our default "Template masking", "AST masking", and "Single-line masking" on bugs in the `Closure` project. Recall that our default setting applies repair templates that we use for cloze-style APR directly on the training data to produce masked lines. AST masking will parse the selected line into an AST and randomly choose a subtree to mask out. On the other hand, single-line masking will simply mask out one entire line in a training sample. We observe that single-line masking performed the worst in terms of the number of correct and plausible patches. This is due to the fact that during repair, we use repair templates that not just regenerate complete lines but also mask out part of the lines. The model just has to regenerate the partial code within the line. Single-line masking is only trained on generating the complete line and thus does not perform well when used with repair templates. Additionally, when compared with AST masking, template masking is able to fix more unique bugs compared with the base CodeT5 since it directly leverages the inference repair templates to create training samples. While AST masking makes use of the structure information, it does not fully emulate the inference setting of cloze-style APR. Furthermore, the two baselines both resulted in a lot of patches with high compilation error rates compared with our proposed template masking strategy. Template masking is able to directly learn the types of repair templates that the final repair task will use as input for the model, resulting in fewer compilation failures. By using template masking for Repair-Oriented fine-tuning, we can train a model that is optimized for the repair task in order to generate more bug fixes to compliment the base CodeT5 model, which shows that fine-tuning model with training strategies that assemble the underlying repair techniques is able to further boost its bug-fixing performance.

3) Impacts of Relevant-Identifier prompting: We examine the different parameters of our Relevant-Identifier prompting strategy. Table VI shows the results of our default (Row Default) approach and other configurations on `Closure`. We first look at the effect of varying the top N identifiers and observe that when only considering the top-1 identifier for each bug we do not generate more correct fixes since it is unlikely that the relevant identifier to fix the bug always has the highest ranking. On the flip side, considering a larger amount of identifiers per bug (top-10, top-20) is also not desirable since we limit the model to sample only 5000 times per bug, and generating more prompts will decrease the number of samples per each prompt. Next, we look at the scope of the project where we find relevant tokens. Our default setting considers only the current file of the bug and we compare this to when we consider the full project (i.e., changing Line 1 of Algorithm 1 to consider all files within the project). We observe that the number of correct and plausible fixes decreases which reflects a similar finding from prior work [39] where a significant amount of correct fixing ingredients (relevant identifiers) can already be found within the same file. Furthermore, by considering the entire project, we could introduce more noise where potentially irrelevant identifiers could be highly ranked. Following, we compare the effect of having type information of the identifier in the prompt. We observe that our default setting (with type) is able to generate more correct fixes compared to without types, indicating the usefulness of such information in helping the model generate the correct usage of the identifier in the patch. Finally, we examine our default prompting method of only providing one relevant identifier at a time. We compare this against another approach to include all the top 5 relevant tokens in the same prompt. We see that separating each relevant identifier to its own prompt provides us with more fixes as including all identifiers together can potentially confuse the model.

TABLE VI: Configurations of Relevant-Identifier

Configuration	#Corr. / #Plaus. (All)	#Corr./#Plaus. (New)
Default	**25 / 39**	**3 / 3**
Top-1	24 / 37	1 / 1
Top-10	23 / 38	1 / 1
Top-20	23 / 37	1 / 1
Full project	23 / 36	1 / 1
No type	24 / 37	2 / 2
Together	24 / 40	1 / 1

4) Overhead of FitRepair: As FitRepair proposes to fine-tune two separate models along with prompting via information retrieval and static analysis, we investigate the extra overhead of using FitRepair compared to just using the base CodeT5. Recall that FitRepair only fine-tunes on the oldest version of the project for Defects4j 1.2 (one-time cost) and uses the trained models to generate patches for all bugs within that project. We find that on average, for each bug in `Closure`, FitRepair adds 14.3 minutes (6.6 for each fine-tuning strategy and 1.0 for prompting strategy compared with directly using the base CodeT5 model. This shows that overall, FitRepair adds a minimal amount of overhead (still within the 5-hour limit including overhead). For practical use, the fine-tuning steps can be done ahead of the actual repair task (e.g., periodically during nights or weekends), incurring no additional time cost compared to previous LLM-based APR tools. Developers can then apply the fine-tuned models together with the base model whenever a bug is detected.

C. RQ3: Generalizability of FitRepair

We further evaluate the generalizability of FitRepair on an additional repair dataset of Defects4j 2.0 containing new bugs

TABLE VII: Evaluation on Defects4j 2.0

FitRepair	CodeT5×4	AlphaRepair	SelfAPR	RewardRepair	Recoder
44	42	36	31	25	11

```
                result.append('K');
-           } else if (contains(value, index + 1, 4, "IER")) {
+           } else if (contains(value, index + 1, 3, "IER")) {
                result.append('J');
```

Bug: *Codec-3*

Fig. 9: Repair-Oriented fine-tuning unique patch

and projects. Table VII shows the number of correct bug fixes on single-line bugs in Defects4j 2.0. We observe that FitRepair is able to achieve the state-of-the-art with the highest number of correctly fixed bugs of 44 (8 more than the best-performing baseline). Unlike other NMT-based or traditional template-based APR tools, FitRepair does not suffer from the dataset overfitting issue of only performing well on the base Defects4j 1.2 dataset. In fact, the relative improvement in the total number of bugs fixed is higher on Defects4j 2.0 (22.2% increase) compared to 1.2 (20.3% increase). Furthermore, comparing against the baseline (Column CodeT5×4), FitRepair is able to improve the number of total bug fixes from 42 to 44 and produce 3 unique bug fixes.

Figure 9 shows a bug (`Codec-3`) fixed by FitRepair but cannot be fixed by any other studied APR tool. The root cause of this bug is an off-by-one error. While this bug looks very simple to fix, one reason previous learning-based APR was not able to provide a correct fix could be the unconventional values of "4" and "3". During the training, NMT-based APR can learn from bug-fixing datasets where it is common to use swap a "0" to a "1" and vice-versa to fix a bug. However, changing a "4" to a "3" can be uncommon in the bug-fixing dataset. Cloze-style APR tool that directly leverages LLMs can also have a hard time on this bug since the change is very small even if a direct repair template can be applied. Since this change is very small, the LLM should not add any additional code other than changing "4" to "3". However, during training one single mask span usually represents multiple different tokens, which may cause the base CodeT5 model to generate more tokens than needed. Using FitRepair and specifically the Repair-Oriented fine-tuning strategy, the fine-tuned CodeT5 can learn such short code generation that usually stems from repair templates such as argument replacement. As such, FitRepair is able to generate this simple patch to fix the underlying bug.

VI. Threats To Validity

Internal. Our manual examination in determining the correct patches from plausible patches is one internal threat to validity. Following common APR practice, the first two authors perform a careful analysis of each plausible patch along with multiple discussions to determine the correctness of a patch. We also released the full set of correct patches produced by our approach for public evaluation [88].

Another internal threat to validity comes from using the CodeT5 model which is trained on open-source GitHub code snippets [89]. This means the training data could overlap with our evaluation repair dataset of Defects4j. To address this, we follow prior work [34] and compute the number of patched functions by FitRepair that also exist in the pre-training dataset. In total, out of the 89 bugs fixed on Defects4j 1.2, 13 of these fixes are part of the original pre-training dataset of CodeT5. This shows that the majority of the correct fixes (76/89 = 85%) do not contain any reference developer patch in the training data. Furthermore, for a fair comparison, if we exclude the 13 bugs whose patched functions overlap with the CodeT5 pre-training dataset following prior work [34], we are still able to achieve state-of-the-art performance on Defects4j 1.2 with 76 total fixes compared to 67 from the best-performing baseline on the remaining bugs. This shows that FitRepair is not simply performing well on the datasets due to the developer reference patches that the model saw during pre-training. Similarly, on Defects4j 2.0, we found that 6 fixed bugs have their reference patch function within the training dataset. Applying the same removal comparison, we still achieve the state-of-the-art result of 38 compared to the best-performing baseline of 30 on Defects4j 2.0. Additionally, we also demonstrate that regardless of the overlap between the training and evaluation datasets, by combining project-specific fine-tuning and prompting strategies, we can further improve the performance of the base LLM. Future work to completely address this threat would need to retrain the CodeT5 model from scratch after removing the overlapping functions.

External. The major external threat to validity comes from our evaluation dataset. In this work, we focus mainly on single-hunk/line repair assuming perfect fault localization, which may not be a practical scenario for APR in practice [90], [91], [92]. We plan to address this by evaluating FitRepair on non-perfect fault localization scenario in the future. Furthermore, the performance achieved by FitRepair may not generalize well to other datasets. To address this, we use two different versions of Defects4j, namely 1.2 and 2.0, and demonstrated that FitRepair is able to achieve state-of-the-art results on both datasets. In the future, we plan to perform more evaluations on other repair datasets across multiple programming languages.

VII. Conclusion

In this paper, we have proposed FitRepair, the first fully automated approach to incorporating domain-specific knowledge with the insights of the *plastic surgery hypothesis* for boosting the performance of LLMs for APR. FitRepair opens up a new dimension for LLM-based APR by using both fine-tuning and prompting to combine the power of LLM with project-specific information. Our evaluation results on the popular Defects4j 1.2 and 2.0 datasets show that FitRepair is able to achieve the new state-of-the-art results in fixing 89 and 44 bugs, respectively. Different from prior APR work on plastic surgery hypothesis, FitRepair is fully automated, effective, and general. Moreover, even partial/imprecise information may still effectively guide LLMs for APR!

Acknowledgements

We thank all the reviewers for their insightful feedback and comments. This work was partially supported by NSF grants CCF-2131943, and CCF-2141474, as well as Kwai Inc.

REFERENCES

[1] E. Richards, "Software's dangerous aspect," *The Washington Post*, 1990, https://www.washingtonpost.com/archive/politics/1990/12/09/softwares-dangerous-aspect/9b2e9243-8deb-4ac7-9e8f-968de0806e5e/.

[2] S. Matteson, "Report: Software failure caused $1.7 trillion in financial losses in 2017," *TechRepublic*, 2018, https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/.

[3] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *SIGPLAN Not.*, vol. 51, no. 1, p. 298–312, jan 2016.

[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[5] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *ESEC/FSE 2015*, 2015, p. 166–178.

[6] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *ISSTA 2020*, 2020.

[7] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *ASE 2018*, 2018, p. 832–837.

[8] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA 2015*, 2015, p. 24–36.

[9] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA 2019*, 2019, pp. 19–30.

[10] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

[11] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE 2018*, 2018, p. 1–11.

[12] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: A tool for automated program repair approach using lazy candidate generation," in *ESEC/FSE 2018*, 2018, p. 888–891.

[13] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *ISSTA 2016*. Association for Computing Machinery, 2016, p. 441–444.

[14] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1980–2024, 2020.

[15] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *ISSTA 2019*, 2019, p. 31–42.

[16] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2019, pp. 456–467.

[17] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[18] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[19] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 765–784, 2013.

[20] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[21] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE 2016*, 2016, p. 691–701.

[22] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[23] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, ser. CSTVA 2014, 2014, p. 30–39.

[24] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *ICSE 2020*, 2020, p. 602–614.

[25] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transaction on Software Engineering*, 2019.

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

[27] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *ISSTA 2020*, 2020, p. 101–114.

[28] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2021, p. 341–353.

[29] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *ICSE 2022*, 2022, pp. 1506–1518.

[30] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *ICSE 2023*, 2023.

[31] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *ICSE 2023*, 2023, pp. 1456–1468.

[32] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014, arXiv:1409.3215.

[33] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *ASE 2007*, 2007, p. 433–436.

[34] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *ESEC/FSE 2022*, 2022.

[35] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE 2023*, 2023.

[36] S. D. Kolak, R. Martins, C. L. Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Deep Learning for Code Workshop*, 2022.

[37] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2022, pp. 69–75.

[38] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *ICSE 2023*, 2023.

[39] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *FSE 2014*, 2014, p. 306–317.

[40] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *ASE 2017*, 2017, pp. 648–659.

[41] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA 2018*, F. Tip and E. Bodden, Eds., 2018, pp. 298–309.

[42] S. J. Yue Wang, Weishi Wang and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *EMNLP 2021*, 2021.

[43] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440.

[44] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021, arXiv:2107.03374.

[45] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022, arXiv:2204.05999.

[46] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*,

ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–10.

[47] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018, arXiv:1810.04805.

[48] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *arXiv preprint arXiv:2304.02014*, 2023.

[49] S. Omri and C. Sinz, "Deep learning for software defect prediction: A survey," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 209–214.

[50] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "Bridging pre-trained models and downstream tasks for source code understanding," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 287–298.

[51] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2022, arXiv:2203.13474.

[52] K. Kuznia, S. Mishra, M. Parmar, and C. Baral, "Less is more: Summary of long instructions is better for program synthesis," 2022, arXiv:2203.08597.

[53] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, arXiv:1706.03762.

[54] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[55] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[56] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[57] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[58] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *arXiv preprint arXiv:2305.01210*, 2023.

[59] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[60] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020, arXiv:2005.14165.

[61] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021.

[62] L. Tunstall, L. von Werra, and T. Wolf, *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

[63] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020, arXiv:2002.08155.

[64] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021.

[65] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," 2020.

[66] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, jan 2020.

[67] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019, arXiv:1910.13461.

[68] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021.

[69] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE22. Association for Computing Machinery, 2022.

[70] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *ASE 2019*, 2019, pp. 255–266.

[71] OpenAI, "Chatgpt: Optimizing language models for dialogue," 2022, https://openai.com/blog/chatgpt/.

[72] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, "Fine-tuning language models from human preferences," 2019, arXiv:1909.08593.

[73] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[74] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," *arXiv preprint arXiv:2301.08653*, 2023.

[75] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[76] A. Wettig, T. Gao, Z. Zhong, and D. Chen, "Should you mask 15% in masked language modeling?" 2022.

[77] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[78] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.

[79] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

[80] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage." 1990.

[81] M. Asad, K. K. Ganguly, and K. Sakib, "Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 328–332.

[82] "Pytorch," 2022, http://pytorch.org.

[83] "Hugging face," 2022, https://huggingface.co.

[84] "Javaparser," 2023, https://javaparser.org.

[85] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[86] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE 2017*, 2017, pp. 637–647.

[87] M. Martinez, T. Durieux, J. Xuan, R. Sommerard, and M. Monperrus, "Automatic repair of real bugs: An experience report on the defects4j dataset," 2015, arXiv:1505.07002.

[88] "Dataset," 2023, https://zenodo.org/record/8244813.

[89] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020, arXiv:1909.09436.

[90] N. DiGiuseppe and J. A. Jones, "On the influence of multiple faults on coverage-based fault localization," in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 210–220.

[91] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Information and Software Technology*, vol. 124, p. 106312, 2020.

[92] Y. Lin, J. Sun, L. Tran, G. Bai, H. Wang, and J. Dong, "Break the dead end of dynamic slicing: Localizing data and control omission bug," in *ASE 2018*, 2018, pp. 509–519.