Aligning the Objective of LLM-based Program Repair

Junjielong Xu[†], Ying Fu[§], Shin Hwei Tan[||], Pinjia He[†*]

[†]The Chinese University of Hong Kong, Shenzhen, China   [§]Chongqing University, China   [||]Concordia University, Canada
junjielongxu@link.cuhk.edu.cn   fuying@cqu.edu.cn   shinhwei.tan@concordia.ca   hepinjia@cuhk.edu.cn

Abstract—Large language models (LLMs) have achieved decent results on automated program repair (APR). However, the next token prediction training objective of decoder-only LLMs (*e.g.*, GPT-4) is misaligned with the masked span prediction objective of current infilling-style methods, which impedes LLMs from fully leveraging pre-trained knowledge for program repair. In addition, while some LLMs can locate and repair bugs in certain functions using the related artifacts (*e.g.*, test cases), existing methods still depend on statement-level fault localization methods to provide a list of buggy hunks for repair. This restriction hinders LLMs from exploring potential patches beyond the given locations.

In this paper, we investigate a new approach to adapt LLMs to program repair. Our core insight is that LLM's APR capability can be greatly improved by simply aligning the output to their training objective and allowing them to refine the whole program without first identifying faulty statements. Based on this insight, we designed D4C, a straightforward prompting framework for APR. D4C can repair 180 bugs correctly in Defects4J, with each patch being sampled only 10 times. This surpasses the SOTA APR methods with perfect fault localization by 10% and reduces the patch sampling number by 90%. Our findings reveal that (1) objective alignment is crucial for fully exploiting LLM's pre-trained capability, and (2) replacing the traditional localize-buggy-hunks-then-repair workflow with direct debugging is more effective for LLM-based APR methods. Thus, we believe this paper introduces a new mindset for harnessing LLMs in APR.

Index Terms—Automated Program Repair, Large Language Model, Objective Alignment

I. INTRODUCTION

Program repair is a critical part of the software cycle. To fix bugs in software systems, developers often need to dedicate substantial time (exceeding 35% of regular development time) for manual program repair [1]. To reduce such human effort, researchers have started exploring Automatic Program Repair (APR) methods [2], [3]. Based on how the patches are generated, APR methods can be categorized into heuristic-based [4], [5], constraint-based [6], [7], template-based [8], [9], and learning-based methods [10], [11]. Traditional methods often rely on pre-defined patterns for solving specific bug types (*e.g.*, NullPointerException). Learning-based methods typically use large-scale, high-quality bug-fix pair data to train neural machine translation (NMT) models [12], which transform a buggy program to a fixed version [3]. Recently, as Large Language Models (LLM) [13] have shown strong code understanding abilities, many researchers have begun exploring LLM-based

APR methods [14], [15]. With broad pre-training across diverse general corpus, LLMs can achieve impressive repair performance through prompt engineering [16] or minimal data fine-tuning [17].

However, current application of LLMs is not well-adapted for program repair due to two major reasons. ***First***, **the inference objective of current approaches is misaligned with LLM's training objective.** LLMs can be classified into *encoder-only*, *encoder-decoder*, and *decoder-only*. The first two are typically trained to predict the masked tokens (*i.e.*, *infilling* [18] or *denoising* [19]), while the third is trained to predict the next tokens (*i.e.*, *completion* [13]). Existing LLM-based methods [14], [20], [21] generally adopt the *infilling*-style method [14] to predict the fixed code at the masked buggy code location during inference. The early work achieved decent results, as the models used are also trained on infilling task [14], [20]. Recently, larger decoder-only LLMs like GPT-4 [22] have shown stronger capability on code tasks [23], and people attempt to directly employ them for infilling-style APR [24]–[26]. However, while the model parameters being scaled up by hundreds of times, the fixed bugs did not even doubled [14], [25], which contrasts with the clear scalability in other tasks [23]. We hypothesize that this is due to decoder-only LLMs are trained for completion rather than infilling, resulting in an objective misalignment between training and inference. Specifically, due to the sparsity of parallel `<masked, denoised>` code corpus in training, these LLMs are less proficient in patch generation on masked buggy code. As noted in recent studies [27], [28], objective misalignment can yield significant sub-optimal performance.

Second, **the current workflow limits LLM from fully exploiting its pre-trained capability.** Existing approaches still adhere to the workflow of first using statement-level fault localization (statement-level FL) tools [29] to obtain a ranked list of potential buggy hunks (*i.e.*, contiguous statements to be modified in patches [30]), and then using APR methods to generate patches by modifying the buggy program at these hunks [2], [3]. However, LLMs already have the capability to fix trivial syntax bugs in their generated code independently by referring to different artifacts (*e.g.*, error report from compilers and virtual machines) [15], [31], [32], illustrating the capability of identifying bugs without given buggy hunks. Considering LLM's superior code comprehension ability, and several studies show that current FL tools may fail to provide
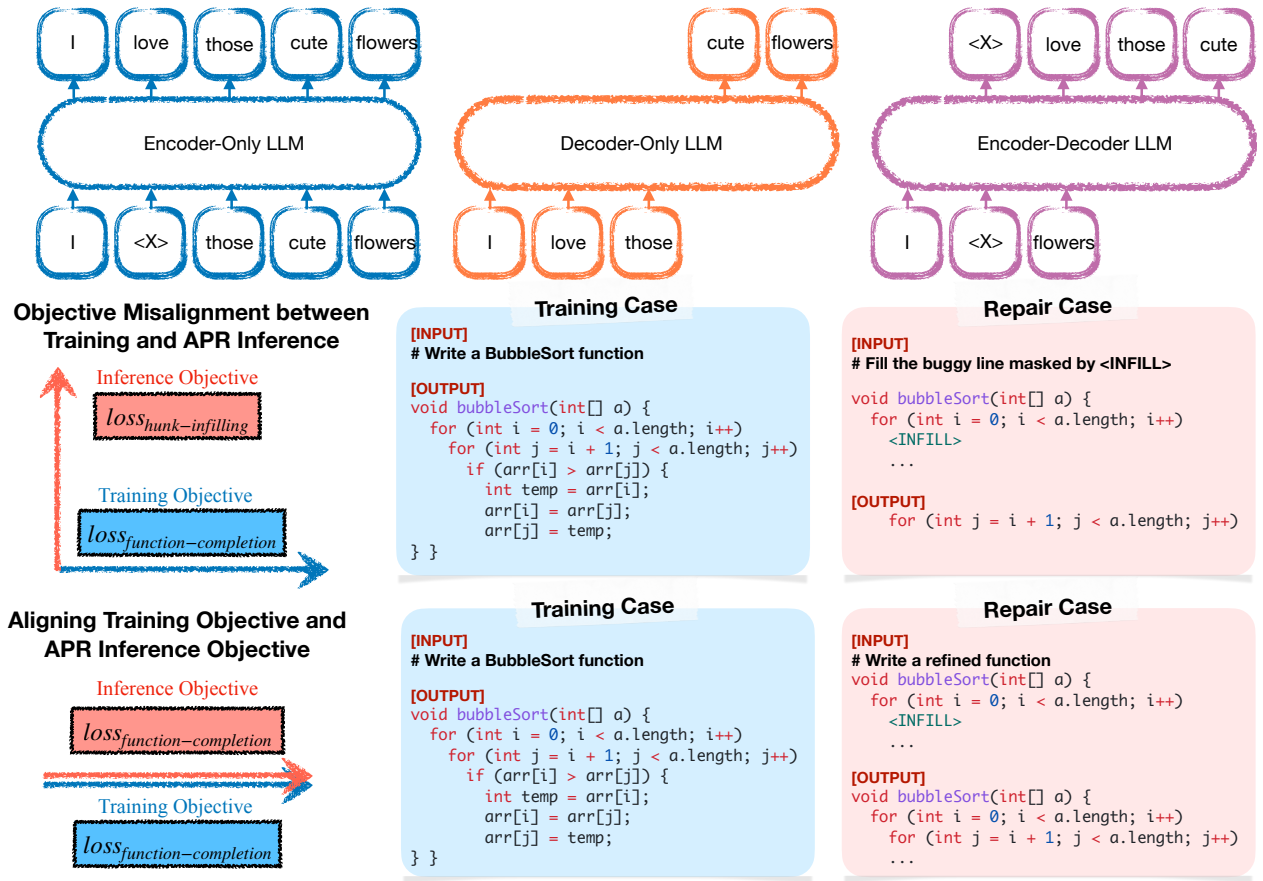
---

*Pinjia He is the corresponding author.

Fig. 1. **First row**: LLM sturctures and their training objectives. **Second row**: The training and inference objective is misaligned when using decoder-only LLMs for infilling-style APR. **Third row**: An intuitive way to align the gap: using LLMs for entire program completion rather than masked span prediction.

precise buggy hunks [33], [34], asking LLMs to generate patches at provided several hunks might contrarily hinder them from exploring broader potential patch space beyond the provided ones. A recent study also revealed that LLMs fail to make good use of the given buggy lines and tend to over-rely on them [17], whereas another study suggested using a more flexible fault localization to enhance APR [35]. As observed in these studies, the traditional localize-buggy-hunks-then-repair workflow may lead to ineffective LLM-based APR.

In this work, we explore a novel way to use LLM for program repair. *We propose that (1) aligning the output from infilling discrete hunks to completing entire functions can better attain the training objective, and (2) allowing LLM to locate and repair buggy hunks with artifacts in a human-like manner can further improve its APR performance.* Based on these insights, we implemented ***D4C*** (*i.e.*, **D**irect **D**ebug **D**rives **D**ecent **C**ode), a straightforward APR approach without complex prompting or extra training. We conducted experiments on a total of 1027 bugs associated with individual functions (*i.e.*, single-function bugs) from Defects4J [36] and DebugBench [37]. D4C repaired 180 out of the 437 single-function bugs in Defects4J, outperforming the state-of-the-art (SOTA) APR methods with perfect statement-level FL [25] by 10%. Meanwhile, each patch in D4C was sampled only 10

times, accounting for 10% used by SOTA methods (which use 100–5000 samples). Our results show that by fully aligning the output format and augmenting the input prompt, D4C can achieve the best APR performance without given buggy hunks, additional fine-tuning, or multi-round dialogue. This paper is not aimed at proposing D4C as a new APR technique, but rather to introduce a new mindset or paradigm for better harnessing LLMs in APR in the future.

In summary, this paper makes the following contributions.

Problem reformulation: We reformulate infiling-style repair problem as a program refinement problem. Our experiments show that asking LLMs to generate an entire refined function can align with the pre-training objective of decoder-only LLMs, leading to SOTA APR performance.

New APR workflow: Instead of relying on statement-level FL tools to obtain the list of buggy hunks for repair as in traditional APR workflow, we show that allowing LLMs to locate and repair buggy hunks by using diverse types of artifacts (*e.g.*, failed tests, error messages, and code comments) in a human-like manner can further improve its APR performance.

Implementation and evaluation: We implemented D4C based on these insights, and the evaluation results on two widely used benchmarks (*i.e.*, Defects4J and DebugBench) have demonstrated its effectiveness.

II. BACKGROUND AND MOTIVATION

A. LLM Architectures and Training Objectives

Background: A large language model (LLM) is a language model consisting of a neural network with many parameters (typically billions of weights or more) trained on large quantities of unlabelled corpus using self-supervised learning [38]. The LLMs usually adopt the Transformer [39] architecture or one of its sub-structures (*i.e.*, *encoder* or *decoder*). The encoder usually consists of feed-forward networks with self-attention [39], while the decoder usually consists of feed-forward networks with cross-attention [39]. Thus, LLMs can be categorized into three types: *encoder-only*, *decoder-only*, and *encoder-decoder* LLMs.

Encoder-only LLMs, such as BERT [18] and its variants like CodeBERT [40], have a bidirectional transformer encoder structure. They are typically trained on the masked language modeling objective (*i.e.*, MLM), aiming to denoise and reconstruct the masked tokens via understanding the surrounding context (Fig. 1). As shown in Eq. 1, the loss of MLM training objective can be explicitly represented as the log-sum of the conditional probabilities of generating the masked token when all the unmasked tokens are known.

$$\mathcal{L}_{MLM} = -\frac{1}{M} \sum_{i \in M} \log P(\hat{t}_i = t_i | t_{k \notin mask}; \theta) \quad (1)$$

where M is the total number of masked tokens.

Decoder-only LLMs, including GPT series [13], [22] and LLaMA series [41], have an autoregressive transformer decoder structure. They are mainly trained on the causal language modeling objective (*i.e.*, CLM), aiming to predict and complete next tokens via following the prefix context (Fig. 1). As shown in Eq. 2, the loss of CLM training objective can be explicitly represented as the log-sum of the conditional probabilities of generating the next token when all the preceeding tokens are known.

$$\mathcal{L}_{CLM} = -\frac{1}{N} \sum_{i \in N} \log P(\hat{t}_i = t_i | t_1, t_2, ..., t_{i-1}; \theta) \quad (2)$$

where N is the total number of the input tokens.

Encoder-decoder LLMs, such as T5 [19] and its variants like CodeT5 [42], have a complete transformer structure. They use an encoder to embed the input text into vector representations, and a decoder to causally generate new tokens after the input prompt. Specifically, T5 series LLMs are often pre-trained on the denoising objective, which aims to generate the reconstructed spans (*i.e.*, a sequence of adjacent tokens) from the masked spans for the masked span in the prompt (Fig. 1). As shown in Eq. 3, the loss of denoising training objective can be explicitly represented as the log-sum of the conditional probabilities of generating the masked token when all input tokens are known.

$$\mathcal{L}_{Denoising} = -\frac{1}{M} \sum_{i \in M} \log P(\hat{t}_i = t_i | t_{k \in input}; \theta) \quad (3)$$

where M is the total number of the masked tokens. While this process is similar to the MLM objective of encoder-only
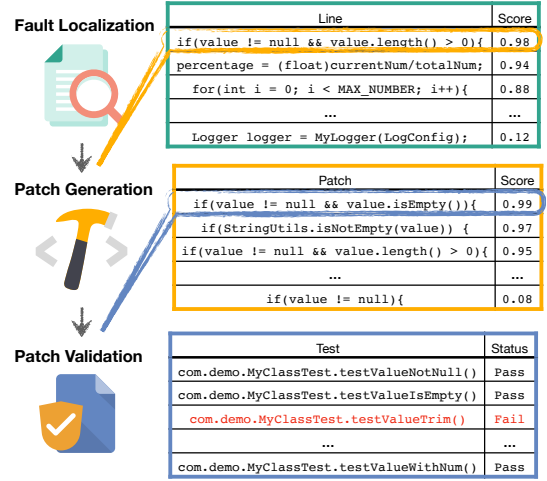

Fig. 2. An example of automated program repair workflow.

LLMs, it only generates the reconstructed tokens from the masked position, ignoring other parts (*e.g.*, "I" and "flowers" in Encoder-Decoder LLM in Fig. 1). Similar to decoder-only LLMs, it also has an unfixed length of text generation.

Motivation: As introduced, both the encoder-only and encoder-decoder (especially T5 series) LLMs involve the corruption and denoising process during pre-training, which substitutes some tokens in the input text for mask tokens, and reconstruct these tokens from the mask tokens. To make effective use of them for APR, Xia *et al.* [14] proposed the infilling-style APR. It aims to replace the entire input buggy hunk with a mask token (*e.g.*, `<INFILL>`) and prompts the model for direct inference, restoring the fixed hunk from the mask. The infilling-style APR can be implemented differently across various models. For instance, in CodeBERT, it recovers the correct code from masked hunks while leaving the rest of the input tokens the same [40]. In CodeT5, it only outputs the correct code restored from the masked hunks [42].

Recently, several APR approaches use decoder-only LLMs as they have exhibited a stronger coding capacity. Influenced by the prior success of infilling, it has been generally assumed that such methods would be useful for decoder-only LLMs. Thus, they use decoder-only LLMs to generate fixed hunks for the given buggy hunks, as illustrated in Fig. 2. However, except for very few models (*e.g.*, InCoder [43]), decoder-only LLMs barely incorporate denoising objective during pre-training. As the `<masked, denoised>` parallel corpus is rarely available, it becomes challenging for the models to learn the generation of the fixed hunk from the masked tokens without further adaptation after the next token prediction pre-training. Such an objective misalignment might hamper LLM's performance on specific tasks extensively, such as the text classification task [28] and the QA task [27], whose inference objectives are misaligned with the training objective of decoder-only LLMs. Thus, we have the following insight.
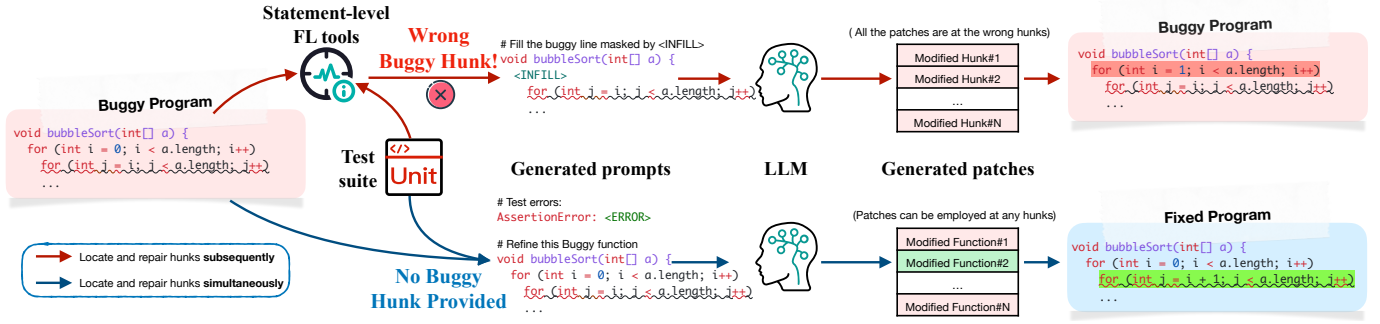
Fig. 3. An example of two different APR paradigms. **First row**: Locate and repair buggy hunks *subsequently* may cause many invalid attempts for patch generation. **Second row**: Locate and repair buggy hunks *simultaneously* can mitigate the cost of patching at specific hunks. (The wavy line is a buggy hunk)

> **Insight 1**: Modifying decoder-only LLMs' output from fixed hunks to the entire refined program can better align the inference objective to the training objective, thus significantly enhancing APR performance.

This insight was inspired by previous approaches focusing on optimizing model performance on specific tasks by aligning the pre-training objectives of the model [27], [28]. We will verify our hypothesis in Sec. IV-C via comparing completion perplexity of two output formats (*i.e.*, fixed hunks or complete function) on white box LLMs (*i.e.*, Mixtral-MoE).

B. APR Techniques and Workflow

Background: Given a buggy program and an artifact (e.g., usually a test suite with at least one failed test), automated program repair (APR) approaches generate a fixed program that fulfills a correctness criteria (e.g., passing all tests). As illustrated in Fig. 2, an APR workflow typically encompasses three steps: (1) fault localization, (2) patch generation, and (3) patch validation. Particularly, APR research mainly focuses on the patch generation step and obtains the fault locations using existing FL techniques (*e.g.*, statistical fault localization) or uses perfect FL results in evaluation.

Based on their patch generation strategies, APR methods can be categorized into heuristic-based [4], [5], [44]–[48], constraint-based [7], [49], [50], template-based [8], [9], [51], and learning-based [10], [11], [52]–[56]. The non-learning-based approaches are usually restricted by a limited set of program transformations, causing the generation of a large number of invalid patches. To identify location to apply these transformations, the current strategy is to first generate a ranked list of suspicious buggy hunks using statement-level FL tools. The APR tool then sequentially generates patches for each provided hunks. Each of the generated patches is then validated using the given artifacts (*e.g.*, test cases). With the advent of deep learning (DL), learning-based APR tools based on neural machine translation (NMT) have emerged. These tools outperform the traditional methods [3] via learning code semantics on a large bug-fix parallel training corpus without specially designed patch templates nor search heuristics. Moreover, since NMT models generally have a limited

parameter size, their training costs and inference efficiency are manageable. Therefore, they are well-suited for the traditional APR workflow.

Motivation: In recent years, there is a growing interest in using LLMs (*e.g.*, GPT-4 [22]) for APR [14], [35], [55], [57]. Researchers usually follow the current APR workflow, positioning LLM as a new patch generator to replace former NMT models. However, taking LLM as a simple substitute in the patch generation step in the current workflow is an under-utilization of its pre-trained knowledge, since these models exhibit the capability of locating and fixing bugs in buggy functions independently. For example, SELF-DEBUGGING [32] has been proposed with the concept of allowing LLMs to progressively refine their generated buggy functions to produce bug-free functions by engaging in multiple self-dialogue rounds with artifacts like execution traces, without using statement-level FL to identify buggy hunks. As shown in Fig. 3, since statement-level FL tools may not always provide perfect predictions of buggy hunks, restricting LLM to fixing the provided hunks may lead to time waste in validating patches at incorrect locations. Moreover, as LLM's inference overhead and time cost is much higher than traditional APR tools, using LLM to generate patches for all given hunks may result in huge resource waste. Thus, we have the insight below:

> **Insight 2**: Prompting LLMs with buggy programs and corresponding artifacts can enable them to locate and repair buggy hunks simultaneously without statement-level FL, thus further improving APR performance.

This insight arises from the APR methodology of programmers who typically identify buggy hunks through tests and documents, and subsequently fix bugs based on these test results. Moreover, existing research revealed that models can produce decent fixes on basic compile bugs when supplemented with compiler errors [15], and their patch generation capability can be further improved with the assistant of the given failed test information [25], [32]. Thus, we can use a more flexible FL method [35] to identify the buggy segments of the program at a coarse-grained level (*e.g.*, using method-level FL [58] to find the buggy function), and allow the LLM
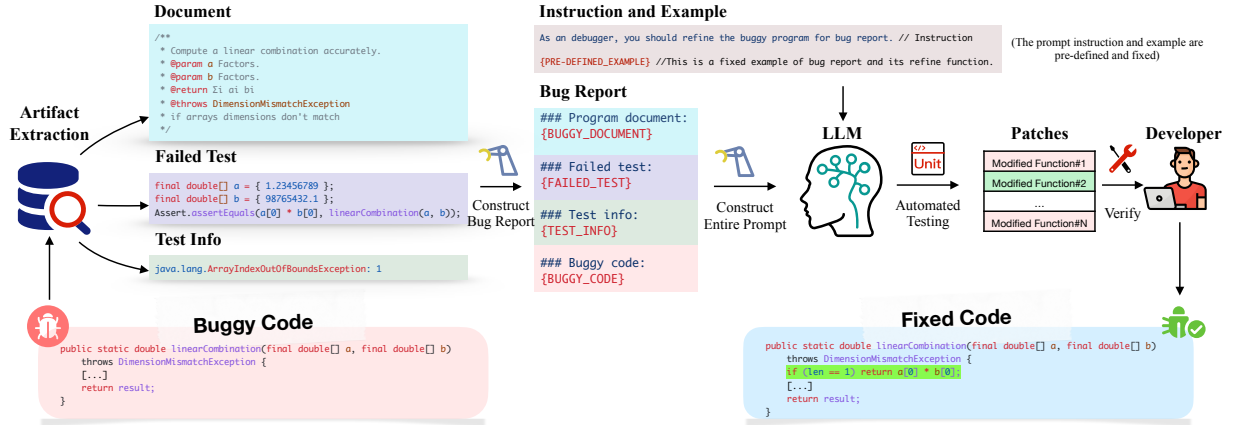
Fig. 4. The workflow of D4C. It uses the buggy code and its corresponding documents, failed tests, and test info (*e.g.*, error message) to construct the prompt for one-shot prompting-based program repair without a specific buggy hunk (usually provided by statement-level FL tools).

to simultaneously locate and fix the buggy hunks within those segments, as locating the segments containing the buggy hunks is easier and more practical than using statement-level FL to directly find the correct buggy hunks for repair. We will verify our hypothesis in Sec. IV-C via comparing the number of correct patches of two input formats (*i.e.*, w/ or w/o artifacts).

III. D4C: DIRECT DEBUG DRIVES DECENT CODE

Based on these two insights, we developed an APR framework D4C, *i.e.*, direct debug drives decent code. As shown in Fig. 4, when presented with a new buggy program[1], D4C uses the related artifacts, including the documents and failed test information, to construct a bug report. Then, D4C will use this bug report to instruct the model to generate a refined version of the buggy program. To align LLM's inference objective with its pre-trained completion objective, we adopt a one-shot prompting strategy. This involves prefixing a fixed example of buggy program and its refined version before the target buggy program in the prompt, enabling LLM to infer their input-output relations and finally generate a refined version of target program in expected format. D4C does not require any additional fine-tuning. It only changes the output format to align with the pre-training objectives, and uses test information and documents (*e.g.*, Javadoc comments) in the input prompt to enable repair without the prior knowledge of the correct buggy hunks. We implemented our approach on GPT-4 series models and Mixtral-MoE models. Next, we will introduce our problem definition (Sec. III-A) and the design details of D4C, including model selection (Sec. III-B), artifact extraction (Sec. III-C), prompt construction (Sec. III-D), patch generation (Sec. III-E) and validation (Sec. III-F).

A. Problem Definition

We model the APR task as a *completion* task that aims at generating a complete refined program based on a buggy

---

[1]In our implementation, due to the LLM's token length limitation, a buggy program refer to a buggy function that can be provided by method-level FL.

program and associated artifacts. This is different from previous LLM-based APR approaches, which treated the APR task as an *infilling* or *cloze* task [14]. Notably, the artifacts are essential for D4C to identify the location to modify, while infilling APR methods rely on statement-level fault localization to pinpoint buggy hunks. Although the artifacts are also used in some approaches [25] to enhance repair, they are not necessarily required by all infilling methods. Specifically, the optimization objective of D4C can be formally written as:

$$\arg\max_\theta \quad P(t_{\text{Fixed Program}}|t_{\text{Buggy Program}}, t_{\text{Artifacts}}; \theta) \quad (4)$$

and the objective of existing infilling-style APR is:

$$\arg\max_\theta \quad P(t_{\text{Fixed Hunk}}|t_{\text{Buggy Program}}, t_{\text{Artifacts}}; \theta) \quad (5)$$

where t means the tokens, θ is the trainable parameters of LLMs. The reason for the difference between D4C and the previous methods in task modeling is that D4C is designed to exploit the pre-training capability of decoder-only LLMs trained on CLM objective, whereas infilling-style APR is designed to exploit the pre-training capability of encoder-only LLMs trained on MLM objective or encoder-decoder LLM trained on denoising objective.

B. Model Selection

The LLM itself is the most vital part of LLM-based applications. Since the goal of proposing D4C is to illustrate an adaptive way to use current advanced decoder-only LLMs for APR, the backbone of D4C should be a decoder-only LLM which has been pre-trained on a substantial code corpus via next token prediction objective. Moreover, D4C does not aim at teaching extra knowledge to LLMs for APR. Instead, its APR ability primarily comes from mining the pre-trained knowledge via aligning the model's response with its training corpus, *e.g.*, a complete function. Thus, to enable D4C exhibit the state-of-the-art APR performance, we choose the most advanced LLMs, GPT-4, to serve as the backbone in our implementation. However, since GPT-4 is a black-box model whose inference loss of CLM objective (*i.e.*, *perplexity*) is

unavailable, we have to choose another state-of-the-art white-box model, Mixtral-MoE, as alternative backbone to validate whether generating a whole refined function can better align the training objective. We select Mixtral-MoE because at the time of submission, the strongest model on the HumanEval leaderboard [59] with the longest context window was Mixtral-MoE. We will not dive into their architecture or pre-training details in this paper, but it is worth noting that the backbone for D4C can be changed to other LLMs with next token prediction (CLM) training objective. To best ensure our replicability, we employed fixed remote API checkpoints or fixed local model versions. For reference, we also provide comparative experiments about using different backbones in Sec. IV-C.

C. Artifact Extraction

Before patch generation, D4C needs to extract related artifacts for subsequent prompting. As shown in Fig. 4, once the buggy function is identified, D4C will automated extract (1) documents or comments that describe the general purpose of the function and its input-output data types, (2) the inputs and expected outputs of the failed test cases, and (3) error messages from executing those failed test cases. Our underlying motivation for providing these artifacts to the LLM is that LLM has gradually exhibited human-like analysis and reasoning abilities recently [31]. Instead of treating LLM as a tool as in the traditional mindset of training NMT models from scratch, we should treat it as an intelligent agent. From this perspective, we contemplate how humans debug, and present this procedure to LLM for execution. Specifically, in the human program repair process, we need to first intuitively understand the function's purpose so we refer to related code documents or function-level comments. Then, we wonder what input can reproduce the bug. Thus, we will check the relevant failed test cases and error messages. Finally, in the absence of communication or confirmation from other developers (*i.e.*, in a *rubber duck debugging* [60] cases), we can only use these artifacts to simultaneously locate and repair the bug. Therefore, we extract artifacts and construct bug report for LLM to repair.

Notably, the implementations of the artifacts extraction procedure may differ due to various data characteristics. For instance, although we default to using function-level comments as documents, if certain functions do not have this artifact, D4C can alternatively exploit related `README` documents. Moreover, if some artifacts are unavailable in some scenarios (*e.g.*, the test cases of the online judgment are unknown), D4C will replace these in the table with a placeholder statement, like "*This program does not possess any known test cases.*"

D. Prompt Construction

After the extraction of artifacts, D4C assembles these artifacts into a comprehensive bug report. As shown in Fig. 4, the report template is concise, with only the most basic notations of the different components in the bug report (*e.g.*, program and comment), without providing any complex instructions or requirements. This allows LLM to focus more on the bug report, thus accomplishing our goal of inducing and exploiting
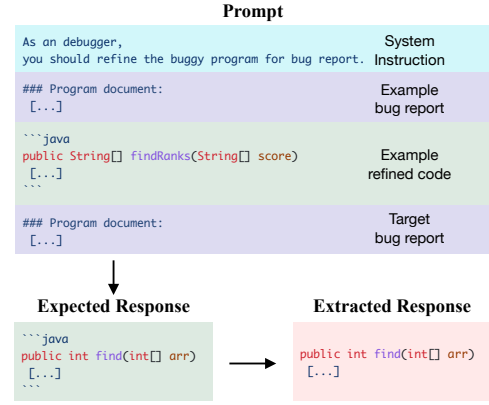

Fig. 5. The prompt structure of D4C. The details of the code are omitted. The example pair is fixed, which is used to constrain LLM's response format.

LLM's pre-trained knowledge and capabilities for program repair, rather than "teaching" the model to repair. Following this, D4C integrates the bug report into a complete prompt, which is carefully designed to include a concise role-play system instruction (*e.g.*, *You are an AI debugger ...*) and a fixed example consisting of a handcrafted bug report and its refined program. Our underlying goal is to stimulate LLM's in-context learning (ICL) ability [13], *i.e.*, analogizing the report-refine pair in the prompt to further understand the instruction. This one-shot ICL also allows us to restrict the output format of the LLM to a *refined function* rather than other common output formats related to APR (*e.g.*, unit diff patch [61]). Moreover, such restricted output format assists us in automatically extracting the corrected functions from LLM response for automated patch verification as illustrated in Fig. 5. We only use one fixed example to make LLMs follow the output format since our goal is using D4C to show the effectiveness of our insight, rather than developing a powerful method with complex prompting strategy like RAG [62], [63].

Notably, the implementation of prompts may vary across different models. Specifically, in the local text completion LLMs, the prompt can be treated as a single text input. In the remote chat models that support multi-turn dialogues, each module of the prompt represents an individual dialogue message in the conversation. For instance, in OpenAI models, instructions are encapsulated within the structure body of system messages, while the input and output of examples are encapsulated within the structure body of user messages and assistant messages respectively. In open-sourced Mixtral, instructions are marked via special tokens `[INST]` and `[/INST]`, while the remaining content is split into different messages using `<\s>` as separators.

E. Patch Generation

Unlike traditional APR tools that generate patches specifically for locations that need modification, the patch generation step of D4C refines the entire program in the inference stage. This is primarily because the core goal of D4C is to explore the extent of APR enhancement by aligning the inference objective

to decoder-only LLM's pre-training objective. Furthermore, we do not introduce an additional patch ranking method, which is widely adopted by current APR approaches. This is because D4C only needs to generate at most *10* patches during decoding (*i.e.*, beam search), which is significantly less than the sampling number of existing methods (*i.e.*, over *hundreds* of sampling). Thus, D4C can subsequently verify all 10 patches without involving significant waste of effort on incorrect patches (within *5 minutes*), rather than suffering from a time-consuming process of verifying hundreds of patches (up to *5 hours* [14], [20], [21], [25], [64]). Additionally, although D4C can rank the patches by comparing their naturalness (quantified by perplexity or entropy) [65], [66], a prior study indicates that it is inaccurate to identify the correctness of the program using naturalness [67]. Therefore, we do not use any re-ranking strategy to save the computational cost incurred by the perplexity calculation. Following previous LLM-based work that focus on method-level generation [68], we assume that the *buggy function* is provided where D4C can select lines within the function to modify. In a real-world scenario, the buggy function can be provided by existing method-level FL tools [58] via analyzing the execution results of failed tests.

F. Patch Validation

After generating the candidate patches, D4C extracts the refined program and uses them to replace the buggy program in the original source code. Then, D4C runs the corresponding test suite to find patches that compile successfully and pass all tests. However, due to the potential incomplete test coverage problem of benchmarks, the APR methods may often produce patches that pass all tests without exactly meeting the expected functionality of the developer [69], thus do not truly fix the bug. These patches are known as *plausible patches*. Ideally, developer validation is helpful, but the bugs in Defects4J are historically fixed and outdated [36], making developer feedback impractical. To ensure rigor experiment, we follow existing APR papers [14], [20], [21], [25], [64] by adding a manual validation for each plausible patch after automated evaluation to identify the correct patches that are the same or semantically equivalent to the human-written patch.

IV. EVALUATION

This section aims to answer the research questions below:

- **RQ1: How does D4C compare against LLM-based APR methods?** We compared D4C to state-of-the-art APR methods with and without perfect FL. We aim to investigate the improvement from aligning output and enhancing input as mentioned in Sec. I.
- **RQ2: How effective are the two insights that drive the design of D4C?** We aim to show that generating a refined function better aligns LLM's training objective, and using the artifacts to locate and repair bugs without given buggy hunks further exploits LLM's APR ability.
- **RQ3: How do different components and parameter affects D4C?** We conduct an ablation study and a sensitivity analysis on D4C. We aim to quantify the

contribution of each prompt component and characterize the performance of D4C in different parameter settings.

A. Experiment Setup

1) Environment and Implementation: We use a black-box LLM (*i.e.*, `gpt-4-0613`) and a white-box LLM (*i.e.*, `mixtral-8x7b-instruct-v0.1`) via OpenAI APIs [70] and 8xA100 NVIDIA GPU server for program repair. We use Python 3.9 to implement the inference and evaluation scripts in a local machine with Ubuntu 20.04.5 LTS. To enhance LLM's instruction following ability, we manually crafted a fixed bug-fix example. This example is employed in each experiment to construct 1-shot prompting for all bugs. To adapt to different experiments, we modified its input-output format according to the corresponding experiment requirement. To control a moderated randomness of text generation, we follow the previous work and set the randomness hyper-parameter, `temperature`, as `1.0`. This is also the default value of OpenAI [70]. To facilitate patch validation, we set a timeout threshold of 1 minutes for each patch. Furthermore, we adopt a much smaller sampling number of `10`, as the inference budgets (fee, time, etc.) for LLMs are much larger than traditional methods. Unless otherwise specified, all experiments adhered to this setting.

TABLE I
STATISTICS OF THE BENCHMARKS: THE NUMBER OF SINGLE-FUNCITON BUGS AND THE THREAT OF DATA LEAKAGE

Benchmark	Bug number	Data leakage	Language
Defects4J	437	Yes	Java
DebugBench	200/194/196	No	C++/Java/Python3

2) Datasets: Our experiments are conducted on 1027 logic single-function bugs from DebugBench (590) and bugs from Defects4J (437). Specifically, DebugBench [37] is a new debug benchmark designed to counter data leakage (by implanting bugs into source data with GPT-4), which contains a total number of 4,253 bugs from Java, C++, and Python3 from LeetCode. In general, fixing logic bug is more difficult than fixing other types of bugs. For example, syntax bugs can be easily detected by the compiler or interpreter without testing, and their fixing methods are usually provided in the raised exceptions. Thus, our paper aligns with the practice of existing APR methods that only focus on fixing logic bugs. Since DebugBench is not exposed to the threat of data leakage, we use it for our insight validation (RQ2). However, due to the complexity to reproduce current baselines on this new benchmark, we do not use it for the comparison between D4C and baselines. Defects4J [36] is a widely used APR benchmark with 835 real-world bugs from 17 open-source repositories. Following previous work [24]–[26], we separate the single-function bugs of Defects4J to v1.2 (203 bugs) and v2.0 (234 bugs). Since it contains data from previous versions of open-source projects where most code models have been trained on, it faces the threat of data leakage. However, this threat is believed to be less severe in APR compared to other

domains [17] due to the sparsity of bug-fix parallel corpus in the training data, meaning that model can only learn the individual buggy or fixed version of the program, not their pairs. Despite this, the potential threat of data leakage could still undermine the validity of hypothesis verification. Hence, Defects4J is mainly used as a reference for comparing D4C with existing baselines. The statistic of the benchmarks are shown in Table I.

3) Metrics: Following the previous work, we use the number of *correct patch* to evaluate the APR effectiveness. Specifically, if a patch can pass all unit tests, then it will considered as a plausible patch. If this plausible patch truly resolves the bug, rather than overfit to pass the unit test only (*e.g.*, via referring the provided failed tests in the prompt), then it will be confirmed as a correct patch. In our evaluation, we adhere to prior studies [14], [25], [64] and manually identify whether a plausible patch truly resolves the bug. Notably, the bugs from DebugBench are collected from LeetCode, and the LeetCode programs that can pass all the unseen test suites from the website are deemed as correct programs. In our experiment, we can only access the *provided test examples* of each LeetCode problem in our bug report, rather than the *failed unseen test* from LeetCode OJ. Thus, we also determine the patches that can pass all the LeetCode tests as correct. To distinguish from manually checked correct patches, we refer to patches can pass the LeetCode validation as *verified patches*.

4) Baselines: In RQ1, we selected several state-of-the-art LLM-based APR methods as baselines, including AlphaRepair [14], FitRepair [21], ChatRepair [25], RAP-Gen [20], and Repilot [64]. We did not compare D4C with learning-based APR methods trained from scratch since our goal is to illustrate the effectiveness of D4C among all the LLM-based approaches. All these baselines employ the infilling objective. In RQ2, we chose different input and output format for comparison to show the superiority of aligning output format and providing artifacts in the input. We also chose prompts that add or delete different information (*e.g.*, function comments) as baselines to investigate the contributions of different information to APR.

B. RQ1: Comparison against LLM-based APR

We compare D4C with five state-of-the-art LLM-based APR approaches on Defects4J, the most commonly used benchmark for APR. Since some of the baselines are not open-sourced, we cannot re-run them to reproduce the result. Thus, we reuse their Defects4J results reported in the original paper. Although using Defects4J may have the the risk of exposing its data to LLM's training corpus, we still use it for evaluation because (1) existing work [17] has shown that data leakage is less of a concern for APR compared to other code-related tasks as the training corpus often contains at most the individual versions of the program (*i.e.*, learning buggy or fixed version only), and LLMs can hardly learn the APR tasks without bug-fix pair corpus during training; and (2) the baselines have also been evaluated on Defects4J (hence, they are subject to the same data leakage risk). Following prior work, we retain this

setting to ensure fairness of the comparison. The comparison result is shown in Table II and Table III.

TABLE II
THE NUMBER OF CORRECT PATCHES GENERATED BY DIFFERENT APPROACHES ON DEFECTS4J V1.2 & V2.0.

Method	Model	v1.2	v2.0	Sum	Sampled
AlphaRepair	CodeBERT	52	34	86	5000
Repilot	InCoder	66	50	116	5000
RAP-Gen	CodeT5	72	53	125	100
FitRepair	CodeT5	89	44	133	5000
ChatRepair	GPT-4	**114**	48	162	100-200
D4C	GPT-4	84	**96**	**180**	**10**

TABLE III
THE NUMBER OF CORRECT PATCHES GENERATED IN PERFECT FL AND STATISTICAL FL ON DEFECTS4J V1.2.

Method	Model	Perf.	Stat.	Drop	Sampled
AlphaRepair	CodeBERT	52	36	30.8%	5000
RAP-Gen	CodeT5	72	48	33.3%	100
D4C	GPT-4	84	**80**	**4.8%**	**10**

In Table II, we calculated the number of correct patches on Defects4J (v1.2, v2.0, and their sum) and patches sampled per bug (notated as "Sampled") in the perfect localization settings. Specifically, for baselines, the perfect *buggy hunks* within the buggy function is provided, while for D4C, only the *buggy function* is provided. Notably, D4C largely outperforms all the existing state-of-the-art methods provided with *perfect buggy hunks* for infilling-style APR. D4C (GPT-4) can generate almost 10% more correct patches than ChatRepair, the latest APR methods which uses GPT-4 for multi-run dialogue with provided buggy hunks. Moreover, D4C only needs to sample 10 times for each bug in patch generation, which is 90% fewer than that required by the most efficient baseline (sampling at least 100 patches). The fewer generated patches also illustrate the better efficiency of D4C and less waste of cost in patch validation. However, using only perfect FL settings does not demonstrate the flexibility and practicality of D4C, which does not rely on given buggy hunks. To illustrate its effectiveness in real-world scenarios, we further conducted experiments of D4C and baselines using existing statistical FL tools.

In Table III, we report the number of correct patches on Defects4J under perfect FL (Perf.) and statistical FL (Stat.) settings. The baselines use statement-level FL tools (*e.g.*, Tarantula [29]) to identify buggy hunks, whereas D4C employs a method-level FL tool (*i.e.*, FLUCCS [58]) to locate buggy functions. We compare D4C only with AlphaRepair and RAP-Gen on Defects4J v1.2 for two reasons: (1) these are the only methods that reported the evaluation results with non-perfect FL setting on Defects4J v1.2 in the paper, and (2) FLUCCS's code supports only Defects4J v1.2. In our experiments, D4C generates only 10 patches for the most suspicious faulty function (Top@1 method) given by FLUCCS. Notably, D4C repairs only 4 fewer bugs compared to the perfect FL
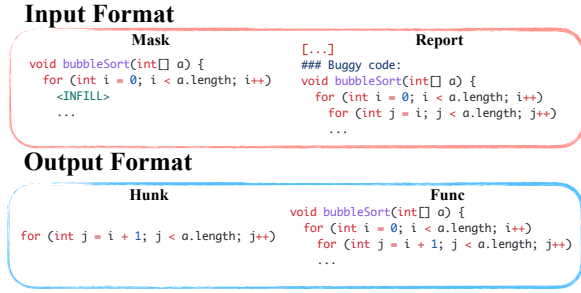
Input Format

Fig. 6. Examples of input and output format used in RQ2.

Format	Perplexity		Mixtral (#Verified)			GPT-4 (#Verified)		
	O	IO	C++	Java	Python	C++	Java	Python
Mask-Hunk	8.59	2.68	66	39	63	140	139	129
Mask-Func	3.01	**1.58**	99	91	76	160	153	147
Report-Hunk	8.50	2.64	72	64	67	144	140	133
Report-Func	**1.39**	1.79	**118**	**125**	**104**	**177**	**163**	**161**

settings, with a minimal drop of 4.8% in repair success. This is significantly lower than the infilling-style APR baselines, which experienced at least a 30% drop. These results suggest that D4C is less affected by inaccuracies in FL since locating buggy functions is easier than locating buggy hunks.

Overall, the results in Table II and Table III suggest that D4C is effective and efficient. However, due to the potential of data leakage, we cannot directly verify our insight of D4C via comparing current approaches. Thus, we conduct more in-depth experiments on DebugBench in Sec. IV-C and Sec. IV-D.

C. RQ2: Effectiveness of the two insights

RQ2 aims to validate the effectiveness of the two insights introduced in Sec. II, and how the two insights contribute to the overall performance improvement in D4C. We design two additional questions for the insights: (IQ1) *Does allowing LLM to generate a complete refined function better align its training objectives and result in better performance?* (IQ2) *Does providing the artifacts enable LLM to locate and repair buggy hunks itself and achieves a better performance?* Specifically, we evaluate the two insights on DebugBench since there is no threat of data leakage. Thus, we use *verified patches* (*i.e.*, patches verified by LeetCode unseen tests) in DebugBench for the evaluation.

1) Validation of Insight 1: To answer the IQ1 question, we calculate the average **perplexity**, the inference loss of CLM objective (Eq. 2), for various output format on the white-box model (*i.e.*, Mixtral-MoE). Measuring perplexity is the most straightforward approach to evaluate whether the generated content aligns with the distribution of training corpus, as LLMs are trained to minimize the CLM objective [13], [41]. Thus, a lower perplexity value indicate the model is more confident at predicting a given sequence. Specifically, we adopt two output formats, including the fixed hunk patches that is used by infilling-style APR (denoted as *Hunk*) and the full-function output used by D4C (*Func*). Fig. 6 shows the examples of these output format. In our implementation, we also carefully modified the system instructions to adapt to each output setting. To control variables, we also calculate the perplexity of generated output (O), and the entire input&output (IO). Thus, we can compare the perplexity among different output format to validate our insight, ignoring the difference between each input format.

Table IV shows the results. We observe that when the output is an entire function, the output perplexity is much lower than using the fixed hunks as the output, regardless of the input. This directly validates that generating a complete function is the better way to align the training objective. To further validate that it is the objective alignment that helps D4C achieve a better APR performance, we evaluate the verified patches generated in each case. The result illustrates that using complete function as output can result in generation of more verified patches with lower perplexity than using discrete hunks. Moreover, when only generating the fixed hunks as the output, the output perplexity is much higher than its perplexity calculated with the whole input and output sequence. This indicates that completing discrete hunks only is misaligned with LLM's training objective. Overall, these findings illustrate the effectiveness of our first insight.

2) Validation of Insight 2: To answer the IQ2 question, we evaluate the patch generation ability on DebugBench using two different input formats, *i.e.*, the bug report (notated as Report) and the buggy program where the buggy hunks are masked (Mask). To control variables, we also conducted experiments using the two previously established output formats as shown in Fig. 6. For the generated patches by *Report-Hunk*, as we do not provide a specific location to modify and cannot precisely determine where the generated patch should be applied, we instruct Mixtral-MoE to help us to automatically apply the generated patches to the original buggy program before patch validation. Table IV shows the number of generated verified patches on both Mixtral-MoE and GPT-4. The results indicate that *Report-Func* is the most effective among all evaluated groups, including *Mask-Hunk*, which is the input-output format of infilling-style APR. This phenomenon aligns with prior evaluations on Defects4J (RQ1). It is worthwhile to note in the experiment in Sec. IV-B, the effectiveness of D4C stems not only from the bug report design but also the objective alignment of the output. However, even when we use the same output format in this experiment, the performance of using bug report as input still outperforms that of using program where the buggy hunks are masked. This suggests that bug report helps not only in locating the buggy hunks, but also give more information for patch generation. Thus, this phenomenon validates our second insight.

Benchmark	Defects4J (#Plausible)			DebugBench (#Verified)			
	v1.2	v2.0	Sum	C++	Java	Python	Sum
w/o Document	93	99	192	175	162	158	495
w/o Test	80	89	169	173	159	159	491
w/o Message	81	93	174	169	162	161	492
- Mask	53	73	126	160	153	147	460
- Pure	47	62	109	143	140	131	414
Samp=1	60	73	133	163	151	145	459
Samp=3	91	88	179	177	163	161	501
Samp=10 (default)	**96**	**114**	**210**	**181**	**171**	**169**	**521**
Temp=0.0	61	75	136	165	154	146	465
Temp=1.0 (default)	**96**	**114**	**210**	**181**	**171**	**169**	**521**

D. RQ3: Ablation study and sensitive analysis

In this section, we aim to quantify the contribution of each component, and investigate the sensitivity of D4C to the sampling number and temperature. We do not additionally validate the sensitivity on different prompts since we have carefully investigated the impact of different prompt formats in RQ2. We use GPT-4 as the backbone, set the default temperature to 1.0, and use the sampling number of 10 by default. As the manual validation of patch correctness on Defects4J is time-consuming, we use the plausible patches in this section. We also use verified patches to evaluate the repair performance on DebugBench. We design the baselines below to check the effectiveness of each component:

w/o Document: D4C without including the program documents or function comments in the prompt.
w/o Test: D4C without including inputs and expected outputs of the failed test cases in the prompt.
w/o Message: D4C without including error message from the failed tests in the prompt.
- Mask: D4C using masks to identify bugs without including any artifacts in the prompt (*Mask-Func*).
- Pure: D4C without including any guidance like mask or artifacts to identify bugs in the prompt, *i.e.*, an aligned LLM.

Table V shows the results. Overall, the results show that each component is important in guiding D4C in generating plausible/verified patches across the two benchmarks. We also observe that the "- Mask" baseline are the least effective among all baselines with bug location guidance because D4C cannot use any information from the artifact to fix the defect. Meanwhile, among the first three baselines where we remove one information from the prompt, we notice that program documents (w/o Document) are the least effective as D4C is able to generate the greatest number of plausible/verified patches without using the document. This is expected as the error message and test are more closely related to the defect than the comments that merely describe general code features. Without guidance on buggy locations, pure GPT-4 can still fix bugs by following the aligned one-shot example pair to generate a refined function. While its effectiveness on

Defects4J might be due to potential data leakage during pre-training, DebugBench is not affected by this issue. Despite this, GPT-4 with the completion objective still fixes more bugs (414 vs 408) on DebugBench than GPT-4 with the infilling objective and masked hunk ("Mask-Hunk" in Table IV). This suggests LLMs have learned to write bug-free code during pre-training. By aligning the inference task with the pre-trained code completion task, we can effectively leverage their pre-trained knowledge to produce high-quality code.

In terms of sample size, we notice that a sample size of 10 is the optimal setting to allow D4C to find the correct patches. We found that increasing the sampling number from 1 to 3 significantly increased the number of plausible and verified patches, but further increasing the sampling number to 10 resulted in a smaller improvement. Since D4C can already achieve satisfactory results when setting the sampling number to 10, we use it as the default sample number for D4C. We also observed that setting the temperature to 0 results in fewer plausible and verified patches. Thus, we adopt a multi-sampling for D4C rather than greedy decoding (temp=0).

V. DISCUSSION

A. Threats to Validity

External Threats. One external threat comes from the potential data leakage problem (*i.e.*, using Defects4J in RQ1, which may have been incorporated in LLM pre-training). We have discussed this threat in RQ1 (*i.e.*, data leakage is not significant on APR [17], and the baselines have also been evaluated on Defects4J). To eliminate this threat in our insight validation, we selected DebugBench, a latest, leakage-free benchmark, for evaluation. This not only keeps a fair comparison in RQ1, but also ensures the credibility of our conclusions in RQ2 and RQ3. As the effectiveness of the model may vary in different settings, our results may not generalize beyond the studied settings and other programming languages beyond the supported ones in Defects4J and DebugBench. We mitigate this threat by reusing configurations in prior work, and widely used benchmarks.

Internal Threats. An internal threat lies in the incomplete test coverage problem [69] of Defects4J, which do not guarantee the correctness of test-passing patches. To mitigate this threat, we follow the previous work [14], [21], [25], [64] to manually check the correctness of plausible patches. To mitigate the potential bias in the manual analysis, two authors independently confirm the patch correctness. Any patches with disagreement were presented to the third author for review. However, since we could not submit the patch to the developers of the corresponding repository for confirmation, this threat could not be completely eliminated. Therefore, we decide to release our experimental results for public verification.

B. Qualitative Analysis of Plausible Patches

Table V shows that D4C generates a total of 210 plausible patches but 30 are incorrect with respect to the human patches. Fig. 7 shows one example of the incorrect patch. Since the failed test contains the `"src"` tag as input, LLM can naturally
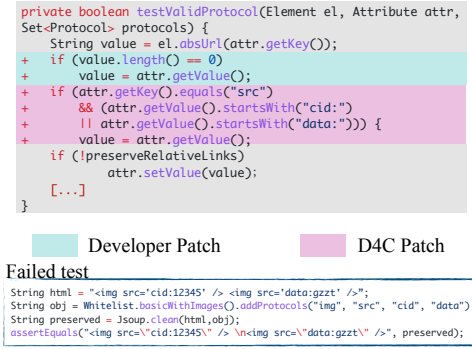
```
private boolean testValidProtocol(Element el, Attribute attr,
Set<Protocol> protocols) {
    String value = el.absUrl(attr.getKey());
+   if (value.length() == 0)
+       value = attr.getValue();
+   if (attr.getKey().equals("src")
+       && (attr.getValue().startsWith("cid:")
+       || attr.getValue().startsWith("data:"))) {
+       value = attr.getValue();
    if (!preserveRelativeLinks)
            attr.setValue(value);
    [...]
}
```

| Developer Patch | D4C Patch |

Failed test
```
String html = "<img src='cid:12345' /> <img src='data:gzzt' />";
String obj = Whitelist.basicWithImages().addProtocols("img", "src", "cid", "data")
String preserved = Jsoup.clean(html,obj);
assertEquals("<img src=\"cid:12345\" /> \n<img src=\"data:gzzt\" />", preserved);
```

Fig. 7. A plausible patch generated at developer patch locations (Jsoup-19)

```
private boolean compute(Object left, Object right) {
    [...]
    double ld = InfoSetUtil.doubleValue(left);
+   if (Double.isNaN(ld)) {
+       return false;
+   }
    double rd = InfoSetUtil.doubleValue(right);
+   if (Double.isNaN(rd)) {
+       return false;
+   }

+   if (Double.isNaN(ld) || Double.isNaN(rd)) return false;
    return evaluateCompare(ld == rd ? 0 : ld < rd ? -1 : 1);
```
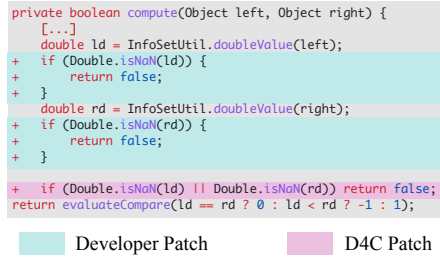
| Developer Patch | D4C Patch |

Fig. 8. A correct patch generated at non-developer patch locations (JxPath-8)

generate patches that only pass the test containing `"src"` without considering patch robustness. This scenario is similar to the overfitting problem in APR [71]. To check for the overfitting problem, two of the authors independently evaluate the patch correctness, followed by a third person to review and confirm their disagreement to ensure the accuracy of the validation. In our manual validation, we also found that when D4C can generate five or more plausible patches for a buggy program in 10 samplings, most of these patches are correct. This observation aligns with previous work (*i.e.*, responses generated multiple times tend to be more reliable [72]). DebugBench does not suffer from the overfitting problem because the given test examples are not used in its unseen test suite when checking for patch correctness.

As we allow D4C to debug without given buggy hunks, we also observe that some correct patches have been applied at different locations compared to the developer patches. Fig. 8 shows an example where the developer patch modifies two if-conditions so it is a 2-hunk bug in the dataset. However, D4C only needs to insert one line, which combines the two conditions and results in a correct patch that is semantically equivalent to the developer patch. Furthermore, if we ask GPT-4 to generate patches at these two hunks, GPT-4 can only write two `Exception` statements, and fails to pass the unit tests. This example shows that allowing LLMs to generate patches without restricting the location may provide more flexibility, allowing generation of correct patches.

C. Token Cost and Price of Patch Generation

The cost of patch generation is a vital criteria to evaluate D4C's practicality. For Defects4J, the average input prompt length is 1,386.84 tokens overall, with 1,325.68 for resolved bugs and 1,369.72 for unresolved bugs. The average output completion length is 314.39 tokens, with 285.98 for resolved bugs and 325.80 for unresolved bugs. Given OpenAI's pricing of $0.01 per 1,000 input tokens and $0.03 per 1,000 output tokens [70], the average cost per patch is $0.023 ($0.021 for resolved bugs and $0.024 for unresolved bugs). Generating 10 patches per bug results in a total cost of $0.23 for the entire dataset. Notably, D4C fixed 180 bugs in Defects4J, with the correct patch being the 4.29th out of 10 patches generated per bug on average (std.=1.74). Thus, if we stop patch generation once a correct patch is found, the average cost per bug will be reduced to $0.18.

D. Wasted Effort of Patch Validation

The wasted effort of patch validation is another criteria to evaluate D4C's practicality. Due to weak test oracles, not all test-passing (plausible) patches are correct [73], and may have anti-patterns [69]. Thus, there is a labor cost of manually validating the correctness of plausible patches beyond the token cost of patch generation. Specifically, D4C generates an average of 5.80 plausible patches per bug (std.=3.28), with every 2.06th plausible patch being correct (std.=1.49). Compared to infilling APR methods which often require users to manually verify over 400 plausible patches before finding a correct patch with patch ranking (and over 600 without ranking) [14], D4C is significantly more user-friendly. Even in the theoretical worst case, users only need to validate up to 10 patches per bug, which is significantly fewer than with existing methods. As discussed in Sec. III-E, this efficiency explains why D4C does not require the additional re-ranking mechanisms to facilitate patch validation. Moreover, there is also a time cost for running unit tests during automated patch validation. However, D4C requires running tests only up to 10 minutes per bug, with a timeout of 1min per patch. This is much fewer than a search time limit of up to 5 hours used in previous work [14], [25]. This illustrates D4C's efficiency.

VI. Conclusion

This paper presents a new approach to adapt LLMs to automated program repair. Our key insight is that the effectiveness of LLM-based APR can be enhanced by (1) aligning the output to their training objective and (2) allowing them to refine the whole program without given buggy hunks. Based on this insight, we designed D4C, an LLM-based prompting framework for APR. Our evaluation shows that D4C outperforms the SOTA APR methods with perfect FL by 10%. Our findings call for adopting a new paradigm and debugging workflow for future LLM-based APR approaches.

VII. Acknowledgment

[1]**Data Availability:** Our source code and experimental results are publicly available at https://github.com/CUHK-Shenzhen-SE/D4C

REFERENCES

[1] D. H. O'Dell, "Understanding the psychology of learning strategies leads to effective problem-solving skills," https://queue.acm.org/detail.cfm?id=3068754/.

[2] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[3] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *arXiv preprint arXiv:2301.03270*, 2023.

[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, 2011.

[5] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1–11.

[6] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[8] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[9] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: a tool for automated program repair approach using lazy candidate generation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 888–891.

[10] 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 Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[11] 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*, 2021, pp. 341–353.

[12] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[14] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[15] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with llms," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140.

[16] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022, pp. 69–75.

[17] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

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

[19] 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.*, vol. 21, no. 1, jan 2020.

[20] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," *arXiv preprint arXiv:2309.06057*, 2023.

[21] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," *arXiv preprint arXiv:2303.10494*, 2023.

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

[23] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[24] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[25] 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.

[26] ——, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[27] D. Cheng, S. Huang, and F. Wei, "Adapting large language models via reading comprehension," *arXiv preprint arXiv:2309.09530*, 2023.

[28] R. Zhong, K. Lee, Z. Zhang, and D. Klein, "Adapting language models for zero-shot learning by meta-tuning on dataset and prompt collections," *arXiv preprint arXiv:2104.04670*, 2021.

[29] 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.

[30] GNU, "Documentation of Diff Hunk," https://www.gnu.org/software/emacs/manual/html_node/emacs/Diff-Mode.html.

[31] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.

[32] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[33] D. Yang, Y. Qi, and X. Mao, "An empirical study on the usage of fault localization in automated program repair," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 504–508.

[34] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[35] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[36] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[37] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Z. Liu, and M. Sun, "Debugbench: Evaluating debugging capability of large language models," *arXiv preprint arXiv:2401.04621*, 2024.

[38] Wikipedia, "Large Language Model," https://en.wikipedia.org/wiki/Large_language_model.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

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

[41] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[42] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[43] 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," *arXiv preprint arXiv:2204.05999*, 2022.

[44] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

[45] G. An, J. Kim, and S. Yoo, "Comparing line and ast granularity level for program repair using pyggi," in *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, 2018, pp. 19–26.

[46] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[47] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[48] 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. IEEE, 2016, pp. 213–224.

[49] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[50] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[51] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 802–811.

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

[53] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[54] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1506–1518.

[55] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[56] T. Ahmed and P. Devanbu, "Better patching using llm prompting, via self-consistency," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1742–1746.

[57] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1506–1518.

[58] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[59] C. Generation, "Code Generation on HumanEval," https://paperswithcode.com/sota/code-generation-on-humaneval.

[60] R. D. Debugging, "Debugging software with a rubber ducky," https://rubberduckdebugging.com/.

[61] GNU, "Documentation of Unified Format," https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[62] J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang *et al.*, "Unilog: Automatic logging via llm and in-context learning," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 129–140.

[63] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Divlog: Log parsing with prompt enhanced in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[64] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," *arXiv preprint arXiv:2309.00608*, 2023.

[65] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[66] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 269–280.

[67] Y. Jiang, H. Liu, Y. Zhang, W. Ji, H. Zhong, and L. Zhang, "Do bugs lead to unnaturalness of source code?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1085–1096.

[68] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 865–865.

[69] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 727–738.

[70] OpenAI, "Documentation of OpenAI API," https://platform.openai.com/docs/introduction.

[71] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[72] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[73] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.