Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT

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

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

Abstract

Automated Program Repair (APR) aims to automatically generate patches for buggy programs. Traditional APR techniques suffer from a lack of patch variety as they rely heavily on handcrafted or mined bug fixing patterns and cannot easily generalize to other bug/fix types. To address this limitation, recent APR work has been focused on leveraging modern Large Language Models (LLMs) to directly generate patches for APR. Such LLM-based APR tools work by first constructing an input prompt built using the original buggy code and then querying the LLM to either fill-in (cloze-style APR) the correct code at the bug location or to produce a completely new code snippet as the patch. While the LLM-based APR tools are able to achieve state-of-the-art results, they still follow the classic Generate and Validate (G&V) repair paradigm of first generating lots of patches by sampling from the same initial prompt and then validating each one afterwards. This not only leads to many repeated patches that are incorrect, but also misses the crucial and yet previously ignored information in test failures as well as in plausible patches.

To address these aforementioned limitations, we propose ChatRepair, the first *fully automated conversation-driven* APR approach that interleaves patch generation with instant feedback to perform APR in a conversational style. ChatRepair *first feeds the LLM with relevant test failure information to start with, and then learns from both failures and successes of earlier patching attempts of the same bug for more powerful APR.* For earlier patches that failed to pass all tests, we combine the incorrect patches with their corresponding relevant test failure information to construct a new prompt for the LLM to generate the next patch. In this way, we can avoid making the same mistakes. For earlier patches that passed all the tests (i.e., plausible patches), we further ask the LLM to generate alternative variations of the original plausible patches. In this way, we can further build on and learn from earlier successes to generate more plausible patches to increase the chance of having correct patches. While our approach is general, we implement ChatRepair using state-of-the-art dialogue-based LLM – ChatGPT. Our evaluation on the widely studied Defects4j dataset shows that ChatRepair is able to achieve the new state-of-the-art in repair performance, achieving 114 and 48 correct fixes on Defects4j 1.2 and 2.0 respectively. By calculating the cost of accessing ChatGPT, we can fix 162 out of 337 bugs for $0.42 each!

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

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680323

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Automated Program Repair, Large Language Model

ACM Reference Format:
Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680323

1 Introduction

Automated Program Repair (APR) [22, 24] is a promising approach to automatically generate patches for bugs in software. Traditional APR tools often use the Generate and Validate (G&V) [44] paradigm to first generate a large set of candidate patches and then validate each one against the original test suite to discover a set of *plausible* patches (which pass all the tests). These plausible patches are then given to the developers to find a *correct* patch that correctly fixes the underlying bug. Traditional APR techniques can be categorized into template-based [23, 26, 40, 41, 49], heuristic-based [35, 37, 67] and constraint-based [16, 34, 43, 50] ones. Among these traditional techniques, template-based APR tools, using handcrafted or mined repair templates to match and fix buggy code patterns, have been regarded as the state-of-the-art [3, 23, 40]. However, template-based tools suffer from lack of patch variety as they cannot easily generalize to bugs and patterns outside of the list of pre-defined templates.

To address the limitations of traditional APR tools, researchers have proposed learning-based APR approaches that leverage advances in Deep Learning. Learning-based approaches are mainly based on either Neural Machine Translation (NMT) [64] or Large Language Model (LLM) [65]. NMT-based APR tools [14, 29, 38, 46, 51, 83, 84] view repair as a translation task to turn buggy code into correct code by training a NMT model [64] using a dataset of historical bug fixes. However, such NMT-based APR tools rely heavily on its training data, obtained by scraping open-source repositories for bug fixing commits. This means that not only can the training dataset be noisy [30] (i.e. containing irrelevant commits/changes) but also that these NMT-based approaches may not generalize to bug fix types not seen in their limited training data.

More recently, researchers have started to directly leverage advanced LLMs for APR [28, 32, 56, 70, 71]. Modern LLMs are trained on billions of open-source code snippets, demonstrating impressive performance on many code-related tasks [6, 12, 21, 74], and can learn to directly generate code given the surrounding context (due to code

Testname: `testZero()`
Failure Line: `assertPrint("var x ='\\0';", "var x=\"\\000\"");`
Error Message: `expected:<var x="\0[00]"> but was:<var x="\0[]">`

```
    switch (c) {
-       case '\0': sb.append("\\0"); break;
+       case '\0': sb.append("\\000"); break;
        case '\n': sb.append("\\n"); break;
```

Figure 1: Example bug fix with original testcase information

naturalness [25, 58]). AlphaRepair [72] proposes the first cloze-style (or infilling-style) APR approach, where the buggy code is removed and an LLM directly predicts correct code given the prefix and suffix context. Recent work has also applied LLM-based APR to autocomplete a single correct line [32] or to directly generate a complete fixed function [56]. A more extensive study [71] has investigated applying larger and different LLM architectures (i.e. generative and infilling) for APR, and demonstrates that LLM-based APR tools can achieve the new state-of-the-art repair performance. Meanwhile, the pipeline for existing LLM-based APR still has the following limitations:

1) Missing test failure information. Current LLM-based tools do not consider the rich information within the original bug-exposing tests. Such information can not only help LLMs understand the *meaning* of the source code under tests but can help and hint with concrete code snippets. Figure 1 shows an example bug fix along with the original test failure information. We see that the fix is to swap the appending string to `"\\000"`. This can be an extremely difficult fix for LLM-based approaches since this unique string is not a commonly used string seen during pre-training and also there are no other examples of triple strings (`"\\XXX"`) within the current function context. However, from the failure line within the test and the corresponding error message, we see that the test expects the output to contain the triple zeros and even contains a code snippet (`"\\000"`) which is directly used in the patch! LLMs have shown powerful in processing/exploiting such unstructured/complex information like test failure logs. By failing to consider them, LLM-based tools may waste a lot of time generating irrelevant patches.

2) Repeated sampling. Current LLM-based approaches first construct an input prompt using the original buggy code and either ask the LLM to fill-in the correct code (i.e. cloze-style APR [72]) or generate a completely new fixed function [56, 71]. Using the initial prompt, LLM-based techniques will sample the LLM multiple times to generate many patches, akin to the traditional G&V paradigm of program repair. However, since each sample is identically independent, the LLM does not know any previously generated patches. As such, LLM-based tools may generate many repeated or similar patches that were already determined to be incorrect, wasting dollar cost in API access or time in GPU execution. Furthermore, this repeated sampling procedure is also drastically different from how human developers fix bugs, where we iterative build on top of the knowledge and tries from previous failed attempts to come up with the next possible patch.

3) Ignorance of valuable plausible patches. In addition to failing to use past incorrect patches, current LLM-based APR tools also cannot effectively exploit the plausible patches generated earlier. Plausible patches have been shown to be valuable since they often share similar locations with the actual correct patches [23, 45]. Moreover, we further hypothesize that plausible patches may also include key code ingredients to pass all tests, and may also help LLMs better learn how to pass all tests to generate more plausible patches (thus increasing the chance of generating correct patches). By ignoring such valuable plausible patch information and starting from scratch after generating plausible patches, existing LLM-based APR may miss opportunities to correctly fix more bugs.

Our Work. We present ChatRepair – a fully automated *conversation-driven* APR approach that interleaves patch generation with instant feedback to perform patch generation in a conversational style. While our idea is general, to build ChatRepair, we use the recently developed, current state-of-the-art dialogue-based LLM – ChatGPT [61][1], which is not only trained on billions of code snippets, but also is designed to be used in a conversational manner to better understand instructions. ChatRepair first extracts relevant test failure information to serve as the initial prompt to provide ChatGPT more contextual information for APR. Moreover, ChatRepair further learns from both failures and successes of earlier patching attempts of the same bug for more powerful APR. For earlier patches that failed to pass all tests, we combine the incorrect patches with their corresponding test failure information to construct a new prompt for the LLM to generate the next patch. In this way, we can avoid making the same mistakes. For earlier patches that passed all the tests (i.e., plausible patches), we further ask the LLM to generate alternative variations of the original plausible patches. In this way, we can further build on and learn from earlier successes to generate more plausible patches to increase the chance of having correct patches. As our approach uses the ChatGPT model, we also compute the dollar cost of ChatGPT API queries used to fix a bug. Surprisingly, we found that *by using ChatRepair, we can fix 162 out of 337 bugs for $0.42 each.*[2]

This paper makes the following contributions:

- **Dimension.** We open a new dimension of conversation-driven paradigm for fully automated program repair and beyond. Our work demonstrates for the first time that we can effectively leverage previously ignored test failure information, as well as earlier patch attempts in a conversational manner to prompt LLMs to generate more correct patches. Moreover, we show the promising future of leveraging dialogue-based LLMs for APR in general.
- **Technique.** We develop ChatRepair, a conversation-driven APR tool using the ChatGPT model. More specifically, we automatically extract concise and relevant information about the initial test failures as well as earlier patch attempts to prompt ChatGPT for effective APR.
- **Evaluation.** We evaluate ChatRepair against current state-of-the-art learning-based and traditional APR tools on the widely studied Defects4j 1.2, 2.0 [31], QuixBugs [39], and ConDefect [69] datasets. ChatRepair obtains the new state-of-the-art repair result of 114 and 48 correct bug fixes (15 and 4 more than prior best baseline) on Defects4j 1.2 and 2.0 respectively. Additionally, we conduct an extensive ablation study to demonstrate the improvement gained from both utilizing rich semantic test failure information and the conversational paradigm of ChatRepair for repair.

2 Background & Related Work

2.1 Large Language Model

Large Language Models (LLMs) [6] have seen meteoric rise in both performance and corresponding adoptions due to recent advances in Natural Language Processing (NLP) that enable scaling LLM size to

[1]While repair uses ChatGPT, no part of this paper is written by ChatGPT.
[2]This is a reference to a prior classic study done for APR [36] not using ChatGPT.

billions of parameters and using billions of training samples. As LLMs are trained to be general and can capture knowledge from various different domains, LLMs are either *fine-tuned* [57] or *prompted* [42] for a downstream task. Fine-tuning involves updating the model parameters with a specific training dataset to target a particular downstream task. However, fine-tuning is not only expensive as it requires additional model training, but may also be infeasible in cases where sufficient training datasets are unavailable. Prompting on the other hand directly uses LLMs without any training by providing natural language descriptions of the downstream task and optionally a few demonstrations of the task being solved as input to the LLM.

LLMs are built on the transformer architecture [65] and can be classified based on the component(s) used. Decoder-only models (e.g., Codex [12] and CodeGen [53]) are the popular GPT-based models trained using Causal Language Modeling objective by training to predict the probability of the next token given all previous left only context. Encoder-only (e.g., CodeBERT [20]) and Encoder-Decoder (e.g., CodeT5 [81]) models are trained using Masked Language Modeling (MLM) or Masked Span Prediction (MSP) objective, respectively, where a small portion (e.g., 15%) of the tokens are replaced with either masked tokens or masked span tokens and the LLMs are trained to recover the masked out tokens based on bi-directional context.

More recently, researchers have proposed LLMs trained using reinforcement learning which *aligns* better with human preference [54, 61, 85]. Examples include InstructGPT [54] and ChatGPT [61], which are first initialized from a pre-trained model on autoregressive generation and then fine-tuned using reinforcement learning from human feedback (RLHF) [85]. RLHF first fine-tunes the base model using a small dataset of prompts (input) and desired output (human-written). Then, a separate reward model is trained on a larger set of prompts by sampling multiple outputs from the fine-tuned LLM and using a human labeler to rank each individual output. Finally, reinforcement learning (e.g., Proximal Policy Optimization [60]) is applied to calculate the reward of the output generated based on the reward model and correspondingly update the LLM weights. The resulting LLM has shown better understanding of complex input prompts and follow instructions to perform various tasks [1, 54, 61]. Specifically, ChatGPT has received lots of attention due to its dialogue/conversation focus by training specifically on conversations and its ability to keep track of and reference prior conversations.

In this work, we continue to build on our in-progress work [73] by introducing a more comprehensive approach that includes more robust feedback and aims to learn from both failing and plausible patches. We not only demonstrate for the first time that LLMs fine-tuned on human preference can be directly applied for APR, but also leverage the instruction and dialogue focus/aspect of these LLMs to build the first fully automated conversation-driven APR approach.

2.2 Automated Program Repair

Automated Program Repair (APR) can help developers by generating patches for a given bug based on its potential fault location(s). Classic APR techniques can be mainly classified as heuristic-based [35, 37, 67], constraint-based [16, 34, 43, 50] and template-based [23, 26, 40, 41, 49] ones. Due to the high number of bugs fixed, template-based APR tools have been recognized as the state-of-the-art. Meanwhile, such APR tools leverage human-defined or automatically-mined templates to first match potential buggy code patterns and then apply the corresponding fixes. However, template-based tools can only fix the bugs that fall into their limited set of patterns and therefore cannot generalize to other bug types or fixes. To address this issue, researchers have proposed learning-based APR techniques by leveraging recent advances Deep Learning. Techniques based on NMT have been extensively studied in recent years, e.g., TENURE [51], Tare [84], SelfAPR [77], RewardRepair [78], Recoder [83], CURE [29] and CoCoNuT [46]. They share the same insight that APR can be viewed as a NMT problem which aims to translate buggy code into correct code. In this way, they can learn to generate patches by training on datasets of pairs of buggy and fixed code snippets. Such NMT-based techniques rely heavily on historical bug-fixing training datasets which are usually obtained from scraping open-source repositories for bug-fixing commits. As such, the training data may include various noises such as irrelevant changes/commits; moreover, in order to reduce such false positives, these datasets focus mainly on small commits which further limit the types of bugs/fixes used for training.

To further combat the limitations of NMT-based tools, researchers have also explored the possibility of directly leveraging LLMs to synthesize correct patches. LLMs, by pre-training on large amounts of open-source code snippets, can directly synthesize the correct code given the surrounding context without having to translate from the buggy code. AlphaRepair [72] is the first tool for cloze-style (or infilling-style) APR where the buggy line(s) is first replaced with masked tokens and then LLMs are used to directly fill-in the correct code based on its context. AlphaRepair shows for the first time that LLM-based APR can outperform the widely studied NMT-based APR techniques on real-world systems. Prenner et al. [56] and Kolak et al. [32] also directly used Codex [12] to generate a fixed function given the original buggy function or to autocomplete a single line given the prefix code on a small dataset (QuixBugs [39]). More recently, Xia et al. [71] conducted an extensive study of LLM-based APR techniques based on various LLMs (e.g., Codex [12], GPT-NeoX [4], CodeT5 [81], and InCoder [21]), and further demonstrated the superiority of LLM-based APR. In addition, researchers have built FitRepair [70], an improved cloze-style APR tool, that leverages the widely known plastic surgery hypothesis [2] by training and prompting using buggy project-specific information to further boost repair performance. Despite the promising results of LLM-based APR, such existing techniques only focus on the source code under repair without considering the rich semantics in test failure information. Furthermore, prior LLM-based techniques continuously sample from the same initial prompt, failing to utilize knowledge from previous failed or plausible patches. In ChatRepair, we address both limitations of prior LLM-based tools by introducing a conversation-based repair paradigm to incorporate both patch generation history with immediate validation feedback to perform repair.

Prior APR tools have also leveraged simple patch execution or test information for APR. GenProg [37] is a classic APR tool that uses an evolutionary algorithm to combine candidate patches that pass more tests together. Constraint-based APR tools [16, 18, 43, 52] have used the underlying testing code to extract and build constraints for patch synthesis. Recently, RewardRepair [78] proposes to train a NMT model with a reward function based on whether a patch in the training set passes compilation or test execution. SelfAPR [77] is another NMT-based APR tool which encodes the bug-exposing

test errors together with the original buggy code as input for APR. Meanwhile, to our knowledge, ChatRepair is the first work that leverages detailed feedback (e.g., including relevant test code and error messages) for each and every patch validated for conversational APR. Also, ChatRepair directly leverages LLMs for digesting test feedback extraction, which is fully automated/generalizable and can understand deep semantic information.

Since the initial version of ChatRepair [73], researchers have proposed to use conversations for APR and other software engineering tasks. Self-Debugging [13] leverages multi-turn LLM feedback to improve program synthesis. Agent-based tools [5, 10, 47, 66, 82], such as SWE-agent [76] and RepairAgent [5], propose to tackle more complex software development tasks by not only using test feedback but also feedback from additional tools (e.g., linters). Furthermore, conversations and the usage of LLMs have been expanded to additional software tasks such as fuzz testing [17, 75], unit test generation [59, 80], and code translation [19, 55].

3 Approach

We propose ChatRepair, a fully automated conversation-driven APR technique that incorporates multiple dimensions of feedback information to iterative query the model to generate patches. Instead of directly generating patches based on the buggy code as existing LLM-based APR techniques do, ChatRepair additionally provides valuable test failure information to further assist LLMs in patch generation. Moreover, instead of continuously sampling from the same prompt as prior LLM-based APR techniques do, ChatRepair keeps track of conversation history and further learns from earlier failed and succeeded patching attempts of the same bug via prompting. In this way, ChatRepair can both avoid prior failures and build on earlier successes (e.g., plausible patches) for more effective APR. As such, ChatRepair maximizes the ability to obtain a genuine correct patch that correctly fixes the underlying bug. While our approach is general and can use different LLMs and be applied to a variety of different repair scenarios, in this work, we use the state-of-the-art ChatGPT model [61] that is designed specifically for dialogue interaction.

Figure 2 shows an overview of ChatRepair using an illustrative repair example. ⚙️ refers to the system message to initialize the model to do a specific task, 🧑 indicates the prompt and feedback ChatRepair provides to the LLM and 🤖 represents the output response given by ChatGPT. First, ChatRepair initializes ChatGPT with the system message of *"You are an Automated Program Repair tool"* to prepare ChatGPT for the repair task. Then, we construct the initial prompt for ChatGPT which contains the buggy function to be fixed and the relevant test failure information to fix the bug (Section 3.1). After querying ChatGPT to generate a potential patch using the initial prompt, we then move onto the conversation stage to first learn from past failures (Section 3.2). More specifically, we evaluate the generated patch against the original test suite to see if the patch can pass the previously failed tests. If not, ChatRepair offers immediate feedback by creating a response that includes the relevant failure information (e.g., test failure/compilation error message) and to re-query ChatGPT to generate a new patch while trying to avoid repeating similar failures. This process is repeated until either a plausible patch is produced or the maximum conversation length is reached. After obtaining a plausible patch, ChatRepair attempts to learn from such successes to generate more plausible patches that pass the test suite (Section 3.3). ChatRepair prompts ChatGPT with earlier plausible patches to generate more alternative plausible patches. From this process, ChatRepair can obtain multiple plausible patches which can increase the chance of getting the correct patch. We next describe each of the steps in more detail.

3.1 Initial Input

To begin with, we use the original buggy project and bug to construct our initial prompt 🧑 to ChatGPT to start off the repair process. We follow prior learning-based APR tools [29, 72, 78] and focus mainly on line-level repair (specifically infilling or cloze-style APR as it has been demonstrated to be the state-of-the-art [72]). Meanwhile, ChatRepair is *general* can also be used in a variety of different repair scenarios (see additional prompts in the top-right of Figure 2), which we will evaluate in more detail during later sections.

Figure 2 shows an example of an initial prompt. Before we add the target bug to be fixed to the prompt, we first include a few examples of historical bug fixes within the same buggy project. By doing so, we gear the model towards the repair task and allow it to learn the desired output format (i.e. a patch) of the task. After the few-shot examples, we take the original target buggy function to be fixed as the input along with the location of the bug. We replace the buggy code within the function with an infill location indicator (⋙ `[ INFILL ]` ⋘) and refer to this later in the prompt to instruct the model to fill-in the correct code. We then provide the original buggy line which we replace with the infill location indicator to the model since the buggy line can also give useful information as to what a candidate patch should look like. Next, we provide additionally relevant information to help ChatRepair to fix the bug. Our approach uses information derived from the failing test(s) which exposes the original bug. Such bug-exposing tests contain rich semantic information/hints which can help with generating the correct patch to fix the bug [45].

ChatRepair uses various information from a failing test, including 1) its name, 2) the relevant test code line(s) triggering the test failure, and 3) the error message produced. The name of the failing test can serve as a *short summary* of the function under test. In the Figure 2 example, the failing test is `testGreatestSubtypeUnionTypes5` which tells us that we are testing for a functionality related to the determining greatest subtype from union types. The relevant test code and error message give concrete information as to why the test failed. In the example, the relevant test code and error message tell the model that we are comparing a `No_OBJECT_TYPE`, but the source code function incorrectly returned `None`. Such failing test information not only offers the model more explanation in terms of the functionality of the source code but also gives concrete information in terms of expected output and function usage to help the model to generate the correct fix. Note, if there are multiple failing tests, ChatRepair only provides the information from one of them to keep a concise initial prompt. Finally, we end our initial prompt by giving the instruction to model to generate a correct line to replace the buggy code at the infill location. Let C be ChatGPT which outputs the probability of generated a sequence, `pre` and `suf` as the prefix and suffix of the buggy code with the buggy line removed, `<infill>` as the special infill token replacing the buggy line, f_0 as the constructed failure test information and I_{fill}

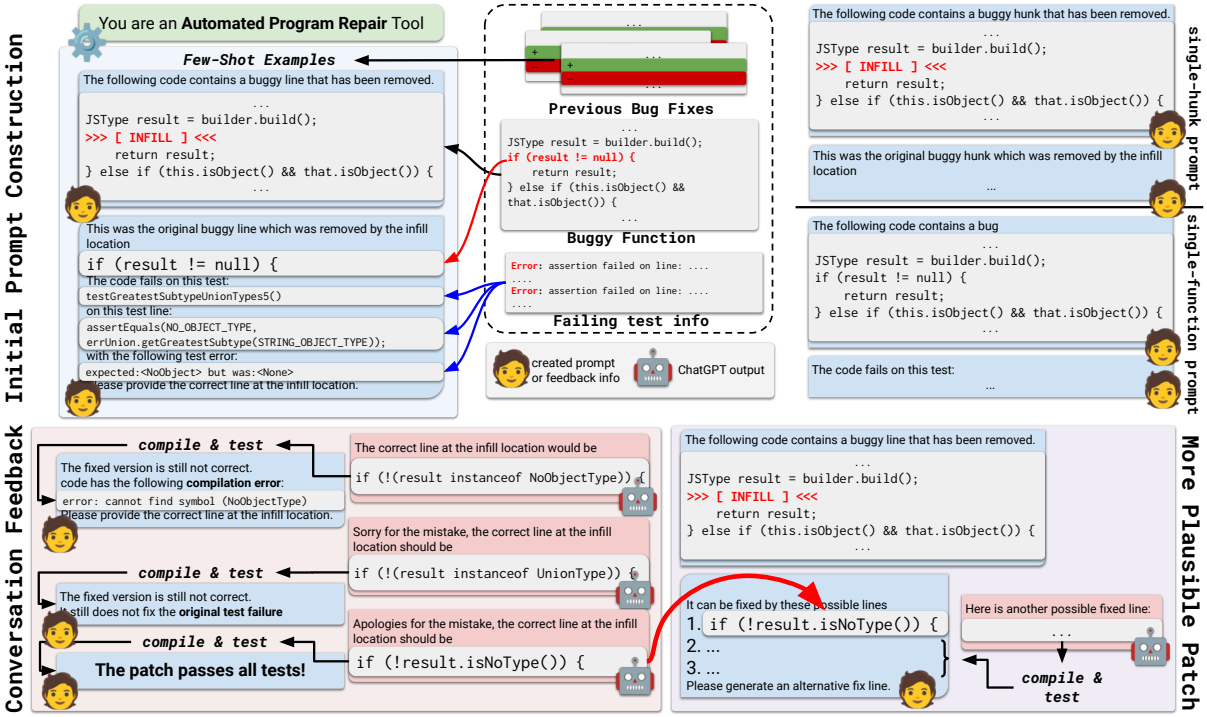


Figure 2: Overview of ChatRepair

as the infill instruction prompt. The patch p generated can be formalized as the conditional probability: $\mathcal{C}(p|\texttt{pre},\texttt{<infill>},\texttt{suf},f_0,I_{fill})$

To our knowledge, ChatRepair is the first work to apply these test failures and error messages in a purely prompting method by combining natural language descriptions of the failure information as input to the powerful ChatGPT model. Different from prior usage of test execution information for repair [77] which relies on custom encodings or handcrafted heuristics, ChatRepair through the use of ChatGPT via prompting is general not only across different programming languages but is also not restricted by the types of test information.

3.2 Conversational Repair

We first use the initial prompt 🧑 created in Section 3.1 to query ChatGPT to obtain a model output 🤖 and extract a candidate patch. Then, we move on to the conversational part where we interleave patch generation with test validation feedback to prompt future generation in a conversational manner. Each generated patch by the model is followed immediately by a patch validation step to compile and run the patch on the test suite. If the patch failed to pass the test, we construct a detailed feedback information using both the incorrect patch and the failing test as part of the prompt for the next patch generation. Similar to the initial prompt, test failure information can help the model understand the failure reason and provide guidance towards generating the correct fix. In conversation step, we further combine test failure information with previously incorrect patches to not only avoid generating more similarly incorrect patches but also learn from the mistakes of prior generations. We repeat the procedure until a plausible patch which passes the entire test suite is generated.

More precisely, we define a **conversation exchange** as a pair of patch generation and validation feedback of that candidate patch (i.e., {🤖, 🧑}). Within one repair conversation, the next patch generated by ChatGPT is prompted with the concatenation of the initial prompt with all previous conversation exchanges. For example, the 3rd patch

Algorithm 1: Conversational Repair

```
1  Function ConversationalRepair:
       Input  : initialPrompt (initial prompt), oFailure (original failing
                test info), testSuite (test suite), ChatGPT,
                maxConvLength (max conversation length), maxTries
                (max tries), AltInstruct (plausible patch prompt)
       Output : pPatches (plausible patches), cost (total cost)
2      pPatches, currentTries, cost ← NONE, 0, $0
3      while currentTries < maxTries and pPatches is NONE do
4          currentLength ← 0
5          input ← initialPrompt
6          while currentLength < maxConvLength do
7              patch, cost ← ChatGPT (input)
8              testResult ← Validate (patch, testSuite)
9              if testResult is PASS then
10                 pPatches ← [ patch ]
11                 break
12             else if testResult is oFailure then
13                 feedback ← "still doesn't fix original failure"
14             else
15                 feedback ← ConstructPrompt (testResult)
16             input ← { input, patch, feedback }
17             currentTries ← currentTries +1
18             currentLength ← currentLength +1
19     if pPatches is not NONE then
20         while currentTries < maxTries do
21             input ← { initialPrompt, pPatches, AltInstruct }
22             patch, cost ← ChatGPT (input)
23             testResult ← Validate (patch, testSuite)
24             if testResult is PASS and patch not in pPatches then
25                 pPatches ← pPatches | [ patch ]
26             currentTries ← currentTries +1
27     return pPatches, cost
```

🤖3 is generated with the input being {🧑$_o$, 🤖1, 🧑1, 🤖2, 🧑2}. Let C be ChatGPT model which outputs the probability of generating a sequence, I be the initial prompt, $Q_{<n}=\{(p_1,f_1),...,(p_{n-1},f_{n-1})\}$ be the previously generated patch p and feedback information f within the same conversation. The next patch generated can be formalized as the conditional probability: $C(p_i|I,Q_{<i})$

Since ChatGPT (and other LLMs) has a limited size context window [8], meaning it cannot take in arbitrary lengthed inputs, we use **conversation length** (i.e., the number of exchanges within a single continuous conversation) as another stopping criteria to restart the repair process from the initial prompt once a maximum conversation length is reached. A maximum conversation length of 1 represents the base case of sampling from the initial prompt over and over again and as we increase maximum conversation length, the amount of history (previous patches/feedback) we provide to the model increases.

Algorithm 1 details our conversation repair process. Our input includes the initial prompt, original test failure information, test suite, ChatGPT model, plausible patch generation prompt (used in Section 3.3), two hyperparameters of maximum conversation length and maximum tries. The final outputs are a list of plausible patches as well as the total cost in ChatGPT API access. Maximum tries is a stopping criteria that stops the repair process once the maximum number of tries (i.e. queries to ChatGPT) has been used to repair a bug (Line 3). Maximum conversation length further limits the prior history used to generate a future patch (Line 6). Following the example in Figure 2, we first set the initial input to ChatGPT as the initial prompt (Line 5). ChatGPT first produces the patch which checks if `result` is an instance of `NoObjectType`. This patch may be motivated by the original test failure information within the initial prompt where a similar global constant (`No_Object_Type`) is used in the test assertion line. However, this patch contains a compilation error. ChatRepair identifies this by directly attempting to compile and run the test suite and constructing a feedback prompt which indicates that the generated patch has a compilation error (`cannot find symbol (NoObjectType)`). To generate the second candidate patch, ChatRepair concatenates the initial prompt, first generated patch, and validation feedback (indicating the compilation error) together as input to ChatGPT.

The second patch indeed fixes the compilation error and checks if `result` is an instance of `UnionType`. ChatRepair employs a **dynamic** feedback approach as described in Algorithm 1. ChatRepair will first compile and run the test suite (Line 8) to observe if the patch can successfully pass the testcases (Line 9). In this example, we see that the patch still fails the original bug-exposing test (Line 12). Instead of repeating the test error message of the original test, we simply refer back to the initial prompt by saying `It still does not fix the original test failure` (Line 13). Since we always include the initial prompt in the input, we can provide a concise message to the model to indicate the test failure reason. On the other hand, if the patch can pass the bug-exposing test used in the initial prompt but fails on a different test (either another original failing test or a regression test), we construct the feedback similar to the initial prompt where we include test name, relevant test code, and error message (Line 15). Note that if there are multiple failing tests, similar to the initial prompt, we only provide feedback for one of them to keep the response succinct.

The third patch is generated similar to before where we concatenate the initial prompt with all previous conversation exchanges. We see that in this case the patch directly calls `isNoType()` is able to successfully pass the test suite. Using the test feedback information such as the error message and the relevant test code, ChatGPT recognizes that this bug deals with a corner case related to none objects or types to generate a plausible patch which fixes the bug.

3.3 Plausible Patch Generation

After the previous step, ChatRepair should obtain a plausible patch that can pass the entire test suite. However, a plausible patch may not always be able to correctly fix the underlying bug since the test suite can be incomplete and therefore not cover all possible intended usage of the underlying code [63]. As such, developers have to manually inspect plausible patches to determine correct ones. Both plausible patches and the final correct patches share the similar characteristic: they all can pass the entire test suite. Therefore, instead of starting from scratch (using the buggy code again), ChatRepair directly leverages the existing plausible patch(es) to create more plausible patches. In short, in order to increase the probability that we can generate a correct patch, ChatRepair takes the plausible patches generated previously, and asks the model to generate alternative variations to produce additional candidate patches.

Figure 2 shows how our plausible patch generation process works. To begin with, we take the initial prompt used (Section 3.1) which contains the original buggy code function along with useful test failure information. We then append the prompt with a list of plausible patches generated (Line 21 in Algorithm 1). In the beginning, this list will only contain the single plausible patch from the previous step, however it grows as we continue to generate additional plausible patches. Next, we indicate in the prompt (AltInstruct in Algorithm 1) of the task we want to solve – `Please generate an alternative fix line.` We then use this prompt as input to ChatGPT and obtain a candidate patch which we will again compile and run the test suite to check if it is indeed another plausible patch (Line 24). We continuously query ChatGPT and update our prompt to include new plausible patches generated in order to avoid repeatedly generating the same plausible patch again and also further build on earlier plausible patches (Line 25). Again let C be ChatGPT model which outputs the probability of generating a sequence, I be the initial prompt, I_{pl} as the task instruction, $PL_{<n}=\{pl_1,...,pl_{n-1}\}$ be the previous generated plausible patch. The next plausible patch generated can be formalized as the conditional probability: $C(pl_i|I,PL_{<i},I_{pl})$

In the end, we obtain a list of plausible patches which can be given to developers for manual inspection. Different from prior APR tools which only operate on the original buggy code to produce patches, ChatRepair leverages additional useful information within each plausible patch to obtain more plausible patches. A plausible patch often contains useful ingredients/patterns that allowed it to pass the original test suite; therefore, instead of starting from scratch (i.e. fixing the bug again), by building on top of existing plausible patches, ChatGPT through its powerful ability to understand instructions can obtain additional plausible patches to increase the likelihood that our final list of patches contains a correct patch that fixes the bug.

4 Experimental Design

We evaluate ChatRepair on the following research questions:

- **RQ1:** How does ChatRepair compare against the state-of-the-art techniques for APR?

- **RQ2:** How does ChatRepair perform when used in different repair scenarios?
- **RQ3:** What are the contributions of different components of ChatRepair in improving repair effectiveness?
- **RQ4:** How does ChatRepair perform on additional recent bugs after ChatGPT training data cut-off date?

We first demonstrate the performance of ChatRepair by comparing against the state-of-the-art APR tools on the popular Defects4j [31] and QuixBugs [39] repair datasets. Following, we closely examine each of our repair scenarios (single-line, single-hunk and single-function) with similarly evaluated baseline tools and also evaluate how our plausible patch generation step helps to improve the number of correct fixes. Furthermore, we conduct a comprehensive ablation study on the different configurations of ChatRepair. In particular, we look at not only the conversational aspect but also how to provide feedback along with the effect on repair performance as we change the maximum length of conversation. Lastly, we further evaluate ChatRepair on additional recently collected bugs to study the performance without potential data leakage affecting the result.

4.1 Implementation

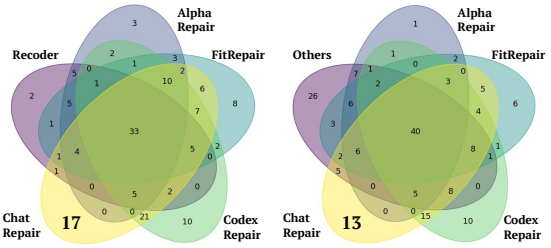
Repair scenarios. In ChatRepair, we study 3 different repair scenarios used in prior work [71]: **single-line**–fixed by replacing/adding a single line, **single-hunk**–fixed by replacing/adding a continuous code hunk and **single-function**–fixed by generating a new function to replace the original buggy version. Our initial prompts differ slightly based on the repair scenario and we provide examples of all three in Figure 2. Note that single-hunk repair setting is studied extensively by prior learning-based APR tools [72, 77, 78].
Implementation. We implement ChatRepair in Python by accessing the ChatGPT API endpoint [7]. We use the `gpt-3.5-turbo-0301` model of the ChatGPT family. For each chosen prompt, the authors follow the best-practice guide [62] and manually examined a few alternative approaches with selected bugs via the Web-version of ChatGPT [9]. We use a sampling temperature of 1 in order to get a diverse set of potential patches. Our default setting for the maximum number of repair attempts allowed (including both initial repair and plausible patch generation steps) is 200 for single-line and single-hunk APR, and 100 for the single-function scenario. We use 1 few-shot example and a maximum conversation length of 3. We evaluate all generated patches on an 8-core workstation with Intel i7 10700KF Comet Lake CPU @3.80GHz and 64GB RAM, running Ubuntu 20.04.3 LTS and OpenJDK Java 64-Bit Server version 1.8.0_312. Following APR work [38, 72, 78, 83, 83], we use a default end-to-end timeout of 5-hours to fix one bug. In reality, our cost is far less than 5-hours due to the low number of patches sampled (<500) per bug.

4.2 Subject Systems

For evaluation, we use the widely studied repair benchmark of Defects4j [31] and QuixBugs [39]. Defects4j is a Java benchmark collected from bug and corresponding fixes of open-source projects. Similar to prior APR tools [23, 71, 72, 77], we separate Defects4j into 1.2 and 2.0. Defects4j 1.2 consists of 391 bugs (after removing 4 depreciated bugs) in 6 different Java projects. In this work, we follow prior study [71] and categorize Defects4j 1.2 into single-function (255 bugs), single-hunk (154 bugs) and single-line (80 bugs). Note that single-hunk is a subset of single-function and single-line is a subset of single-hunk bugs. We then apply our 3 proposed repair scenarios corresponding to the each of the 3 datasets. Note in RQ1, similar to prior work [71] we report the total number of bugs fixed when combining all three repair scenarios together and study each repair scenario separately in later RQs. Defects4j 2.0 consists of 438 new bugs across 9 additional projects. We select only the 82 single-line bugs within Defects4j 2.0 which is the main setting used in prior APR tools for ease of comparison [72]. Additionally, we evaluate on the QuixBugs [39] dataset which is made up of 40 buggy and fixed versions of classic programming problems in both Python and Java. All 40 bugs in QuixBugs-Python are single-function, single-hunk and single-line bugs while 40, 37, and 36 bugs in QuixBugs-Java are single-function, single-hunk and single-line bugs respectively. Lastly, we also evaluate on the recently introduced ConDefect [69], a collection of single-line programming contest bugs from 645 tasks in Java and Python published after October 2021 – the training data cut-off date of ChatGPT to avoid the data leakage issue. We perform further filtering to remove any duplicated buggy programs or tasks and obtain 321 and 330 Java and Python bugs respectively. We use ConDefect as a benchmark to evaluate ChatRepair and baseline techniques without having to worry about data leakage as it only contains problems and bugs created after the knowledge cut-off date of ChatGPT.

4.3 Compared Techniques

Baseline techniques. We compare ChatRepair against current state-of-the-art traditional, NMT learning-based and LLM-based APR baselines. We select 9 recent learning-based and LLM-based APR baselines: FitRepair [70], SelfAPR [77], AlphaRepair [72], RewardRepair [78], Recoder [83], CURE [29], CoCoNuT [46], DLFix [38] and SequenceR [14]. In particular, AlphaRepair and FitRepair are LLM-based repair tools by applying pre-trained CodeBERT [20] and CodeT5 [81] model respectively using cloze-style APR. Furthermore, we also include an LLM-based APR tool built using the Codex model [20] (we refer to as CodexRepair) in a recent study where researchers directly applied LLMs for APR without any fine-tuning [71]. CodexRepair is also studied on three repair settings used in our work which allows for more direct comparison. For traditional APR tools, we compare against 12 selected representative techniques: TBar [40], PraPR [23], AVATAR [41], SimFix [27], FixMiner [33], CapGen [67], JAID [11], SketchFix [26], NOPOL [16], jGenProg [48], jMutRepair [49], and jKali [49]. Altogether, we compare against 22 prior APR tools. Moreover, we also evaluate against a baseline of directly sampling using the ChatGPT model to perform repair without any conversation or feedback information. We refer to this baseline as BaseChatGPT. Since our 3 repair scenarios rely on knowing the location of the bug, we use the perfect fault localization (where the groundtruth location of the bug is given) results from prior tools. This is the preferred evaluation setting as it eliminates any differences in performance caused by running fault localization tools [68, 71, 72, 77, 78, 83]. Following convention in APR work [72, 83], we directly report the fix results obtained in prior studies [23, 71, 77]. As ConDefect is recently collected and produced, there are no reported experimental results for any APR tools. Therefore, we apply state-of-the-art AlphaRepair to ConDefect for baseline comparison.



(a) Learning-based APR tools (b) All APR tools

Figure 3: Bug fix Venn diagram on Defects4j 1.2

Metrics. We use the standard metrics of *plausible patches* – passing the entire test suite and *correct patches* – semantically or syntactically equivalent to the reference developer patch. We follow common practice in APR and manually determine the semantic equivalency to compute correct patches. Additionally, we use metric of *tries* which indicates the number of samples used to obtain either a plausible or correct patch when querying ChatGPT. A lower number of tries is desirable as it reduces the time it takes to fix a bug. Finally, we also compute the *dollar cost* of fixing a bug. At the time of writing, ChatGPT costs $0.002 per every 1000 tokens [7] processed or generated.

5 Evaluation

5.1 RQ1: State-of-the-Art Comparison

We first compare ChatRepair against the state-of-the-art APR tools. Table 1 shows the number of bugs fixed on Defects4j 1.2 and 2.0 by the top baseline tools as well as BaseChatGPT– only using ChatGPT without any test failure information and conversation. We first observe that ChatRepair can improve over the baseline of just using the ChatGPT model with 34 and 23 more bug fixes on Defects4j 1.2 and 2.0 respectively. This improvement is obtained by successfully leveraging the conversational aspect of ChatGPT model to provide immediate feedback using both previous incorrect or plausible patches and test failure information. Interestingly, we also observe that prior tools such as AlphaRepair and FitRepair, which uses much smaller LLMs (CodeBERT and CodeT5), can perform better than CodexRepair in cases like single-line repair on Defects4j 2.0 due to the use of repair-specific templates compared with pure code infilling. In fact, ChatRepair demonstrates for the first time that LLM-based APR without any repair templates can achieve top performance on Defects4j. In total, *ChatRepair is able to achieve 114 and 48 correct bug fixes on Defects4j 1.2 and 2.0 respectively, with 15 and 4 more than the current state-of-the-art APR tools.* Calculating the total cost of query ChatGPT, we can *fix 162 out of 337 bugs for $0.42 each!* While the 114 and 48 fixes are achieved by combining three repair settings together, ChatRepair still generates far less patches (<500 in total per bug) compared to prior learning-based tools which can generate up to 10,000 patches per bug [29, 46]. Similarly in Table 2, ChatRepair is able to correctly fix all bugs within the QuixBugs-Java and -Python datasets, beating out all top-performing techniques.

Figure 3a and 3b shows the Venn diagrams of the bug fixed by top-performing learning-based APR tools and all studied baselines on Defects4j 1.2. We first compare ChatRepair against top-performing learning-based APR tools in Figure 3a and then we select the 3 top baselines in terms of the number of bugs fixed and group all other studied APR tools (not just the top-performing ones in Table 1) as

Testname: `testCreateNumber()`
Failure Line: `0xFADE == NumberUtils.createNumber("0Xfade").intValue()`
Error Message: `0Xfade is not a valid number.`

```
    }
-   if (str.startsWith("0x") || str.startsWith("-0x")) {
+   if (str.startsWith("0x") || str.startsWith("0X") ||
+       str.startsWith("-0x") || str.startsWith("-0X")) {
        case '\n': sb.append("\\n"); break;
```

Figure 4: Unique bug fixed in Defects4j 1.2

Testname: `testNonFiniteDoublesWhenLenient()`
Failure Line: `jsonWriter.value(Double.NaN);`
Error Message: `Numeric values must be finite, but was NaN`

```
    writeDeferredName();
-   if (Double.isNaN(value) || Double.isInfinite(value)){
+   if (!isLenient() && (Double.isNaN(value) ||
+       Double.isInfinite(value))) {
        throw new IllegalArgumentException("Numeric
```

Figure 5: Unique bug fixed in Defects4j 2.0

"Other" in Figure 3b. We see that not only can ChatRepair achieve more unique bugs fixed compared to learning-based baselines but *ChatRepair can provide the correct patch for 13 unique bugs that no prior approach is able to fix so far on Defects4j 1.2.* To illustrate the power of ChatRepair, we show an example bug (`Lang-16`) in Defects4j 1.2 that is only fixed by ChatRepair in Figure 4. The fix is to append two additional conditions of starting with either `"-0X"` or `"0X"` referring to hexadecimal representation of a number. This bug is difficult to fix, since the strings are not commonly found in either bug-fix training data (NMT-based) or in pre-training data (LLM-based). In order to generate these condition, the APR tool needs to understand the expected behavior and what other usage inputs may look like. In fact, one of the condition (`"0X"`) is directly used in the failing test where the test tries to create a number of `"0Xfade"`. ChatRepair is able to leverage this relevant test code information and generate the string used in the test as a condition. Furthermore, the new negative variant `"-0X"` can also be easily generated by ChatRepair as ChatGPT is able to learn from the original buggy line which also contains pairs of negative and positive conditions. Combining both conditions together, ChatRepair is able to obtain the correct patch that fixes this bug.

Another bug (`Gson-15`) that can only be fixed by ChatRepair is presented in Figure 5 from Defects4j 2.0. The fix requires another unique condition of `!isLenient()`. To make things worse, this usage of the function is not found within the original buggy function context. As such, it can be extremely difficult for prior learning-based APR tools to fix since there are no example usages of the condition within the context. However, we observe that the failing test is named `testNonFiniteDoublesWhenLenient` where the word lenient directly appears. ChatGPT, through looking at the failing test name, can understand the semantic meaning of the test which in this case is to test a particular setting with `lenient = true` and generate the correct fix line to check for this unique setting.

5.2 RQ2: Repair Scenarios

Next, we take a look at each of our three repair settings (single-line, single-hunk and single-function) in more detail. For this section, we focus our analysis against BaseChatGPT using ChatGPT without any test failure information or conversation, and CodexRepair which is the best performing LLM-based APR and has also been evaluated on the three repair settings that we use.

Table 1: Correct fixes on Defects4j

Dataset	ChartRepair	BaseChatGPT	CodexRepair	FitRepair	AlphaRepair	SelfAPR	RewardRepair	Recoder	TBar	CURE
Chart	15	9	9	8	9	7	5	10	11	10
Closure	37	23	30	29	23	19	15	21	16	14
Lang	21	15	22	19	13	10	7	11	13	9
Math	32	25	29	24	21	22	19	18	22	19
Mockito	6	6	6	6	5	3	3	2	3	4
Time	3	2	3	3	3	3	1	3	3	1
D4J 1.2	**114**	80	99	89	74	64	50	65	68	57
D4J 2.0	**48**	25	31	44	36	31	25	11	8	-

Table 2: Correct fixes on QuixBugs

QuixBugs	Chat Repair	Base ChatGPT	Codex Repair	Alpha Repair	CoCoNuT
Python	**40**	**40**	**40**	27	19
Java	**40**	**40**	38	28	13

Table 3: Correct fixes using three repair settings

Tools	D4J 1.2			Quixbugs-Py			Quixbugs-J		
	SL	SH	SF	SL	SH	SF	SL	SH	SF
ChatRepair	**57**	**79**	**76**	**39**	**40**	**40**	**36**	**37**	**39**
BaseChatGPT	41	55	45	38	37	35	33	36	**39**
CodexRepair	39	62	63	**39**	39	37	34	34	32

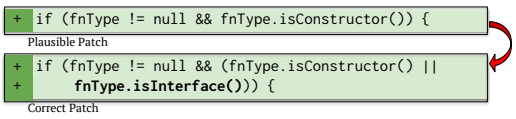


Figure 6: Plausible generation example

Table 3 shows the results of ChatRepair against the two baselines on Defects4j 1.2 and two QuixBugs datasets. Interestingly, we first observe that the base ChatGPT model performs even slightly worse than CodexRepair on the real-world benchmark of Defects4j 1.2. We theorize that this is because ChatGPT is not designed or directly fine-tuned for code generation like Codex. As such, directly using ChatGPT in a similar fashion to prior LLM-based APR tools that solely sample from the same initial prompt without additional information does not yield impressive improvements [79]. On the other hand, by using ChatRepair, which combines the powerful dialogue/instruction understanding ability of ChatGPT with dynamic feedback, ChatRepair is able to better leverage the previously ignored test failure information and earlier patch attempts to better perform APR. Codex on the contrary, is designed mainly for code completion and lacks the ability to be used in a conversational manner. In summary, for each individual repair setting, ChatRepair is able to achieve the highest number of bugs fixed compared to both state-of-the-art CodexRepair and running base ChatGPT.

Additionally, the improvement in more correct fixes does not only come from the conversational and validation feedback aspect but is also contributed by our plausible patch generation step. Recall that once a plausible patch is generated, we directly use that patch to generate more plausible patches by asking ChatGPT to provide other variations of the patch. In summary, plausible patch generation is able to add on average an additional 9.4, 16.6, 5.5 plausible patches, and improve the number of correctly fixed bugs in single-line, single-hunk, and single-function repair scenarios by 4, 7, 2 respectively on Defects4j 1.2. This improvement demonstrates the usefulness of our proposed approach in leveraging the important information in plausible patches to generate more patches leading to a correct fix.

Figure 6 shows an example of a correct fix (`Closure-125`) by ChatRepair which was initially only plausible and then became correct after guiding ChatGPT to learn from the earlier plausible patch. We see that the initial plausible patch produced by ChatRepair is indeed able to pass the developer tests by checking if `fnType` is a constructor. However, the testsuite does not cover all corner cases and the actual correct fix involves checking an additional condition of an interface. By using the plausible patch generation, ChatRepair does not have to start from scratch (using only the buggy code) but instead can build on the knowledge already obtained in the first plausible patch. In this bug fix, ChatRepair adds the additional condition required to correctly fix by learning from the original plausible patch.

5.3 RQ3: Configurations of ChatRepair

We investigate the different configurations of ChatRepair. Specifically we examine the important parameters of (1) initial prompt used, (2) feedback response provided and (3) maximum conversation length. Due to the substantial cost of invoking the ChatGPT API multiple times for each dimension of our ablation study, we focus on the 80 single-line bugs within Defects4j 1.2. Also, we analyze the number of plausible fixes produced instead of correct fixes in this RQ due to the intensive manual efforts involved in patch inspection. Each of our ablation experiments uses the default setting described in Section 4.1 except we use zero-shot (not providing any prior bug fix examples) by default since it can best illustrate the effect of individual components and make it easier for studying the impact of few-shot examples.

Table 4: Initial prompt variations

Initial Prompt	#P	Avg. # tries	Avg. $
BasePrompt	55	22.53	$0.069
TestName+ErrMsg	59	22.47	$0.072
TestName+ErrMsg+FailLine	**64**	**21.86**	**$0.061**
TestName+ErrMsg+TestBody	61	23.42	$0.083
`You are a helpful assistant`	**64**	24.17	$0.074
`You are an APR tool`	**64**	**21.86**	**$0.061**
0-shot	64	21.86	**$0.061**
1-shot	**65**	9.91	$0.072
2-shot	**65**	**9.87**	$0.085

5.3.1 Initial Prompt. In addition to our default initial prompt given to ChatGPT, we also evaluate several alternative variations. Each variation attempts to illustrate some key aspects of information which can be helpful for ChatGPT during the repair process. Table 4 shows the results of the different initial prompts. Row **BasePrompt** refers to the prompt where we only indicate the code contains a bug and asks

the model to provide a fix, **TestName+ErrMsg** includes both the failing test name (e.g., `testGetCategoryIndex`) and test failure error message (e.g., `NullPointerException`), **TestName+ErrMsg+FailLine** additionally includes the exact line where the failure occurred within the test (e.g., `assertEquals(-1, empty.getCategoryIndex("ABC"));`) and **TestName+ErrMsg+TestBody** additionally uses the entire failing test function body instead of just the failure code line.

First, we observe that the base initial prompt of only providing with the buggy code and asking it to generate a patch performs the worst in terms of the number of bugs fixed. We see that by adding auxiliary information such as failing test name and error message, we can further improve the repair performance. Tests that are well named can provide semantic meaning of the test. Error messages also offer unique insights regarding the nature of the test failure (e.g., null point exception, array out of bound checks, etc) and can directly motivate a potential correct patch. Furthermore, we remark that providing the exact line within the test where the failure occurred can also improve repair performance. Such lines may include assertions showing desired results (e.g., numerical comparison) or statements that triggered the exceptions or crashes (e.g., field dereferences). This additionally gives concrete hints to ChatGPT on how to fix the specific bug. Moreover, we observe that the prompt which includes the entire test function code also performs well in terms of the number of bugs fixed. However, on average it costs the most compared to the other initial prompts used. This is because the model incurs additional cost to process the entire test function code for each repair attempt, which can be largely depending on the size of the test function and may contain test code irrelevant to the bug. As such, a more concise prompt which includes just the failing test code line can already achieve effective repair performance while being economic.

Table 4 also shows several other parameters of the initial input apart from the included test failure information. Row **`You are a helpful assistant`** uses the default system message (Section 3) by ChatGPT and **`You are an APR tool`** (we use the full name of APR) is our modified system message. While the number of fixes is similar between the two system messages, we observe that by aligning the system message with the task we want to solve – program repair, the model can arrive at the plausible patch faster (less tries) since it can faster understand the task it is trying to solve. As such we can reduce the cost of ChatRepair by designing specific system messages. Furthermore, we evaluate the effect of having few-shot examples of bug fixes before the target buggy code input in Table 4. We observe that by providing ChatGPT with some examples of prior bug fixes, we obtain a slight increase in number of plausible patches while at the same time drastically reducing the number of tries used to fix the bug. Few-shot examples, similar to the system message, can get the model familiar for bug fixing by understanding the task and input/output formats.

Table 5: Feedback response variations

Feedback Response	#P	Avg. # tries	Avg. $
BaseFeedback	58	23.12	$0.071
TestName+ErrMsg	61	22.48	$0.073
TestName+ErrMsg+FailLine	62	24.71	$0.074
Dynamic	**64**	**21.86**	**$0.061**

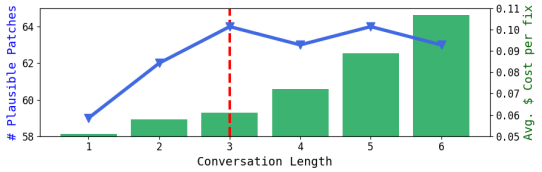


Figure 7: Effect of maximum conversation length

5.3.2 Feedback Response. Another important aspect of ChatRepair's design is the feedback response we provide to the model. Similar to the initial prompt design, we also consider multiple different ways we can provide feedback to the model. Table 5 shows the results of the different feedback response variants. Row **BaseFeedback** means we only tell the model that the generated patch is not correct without any additional feedback. Similar to the initial prompt construction, **TestName+ErrMsg** includes both the failing test name and test failure error message, **TestName+ErrMsg+FailLine** additionally includes the exact line where the failure occurred within the test. Different from initial prompt construction, **Dynamic** is our default approach where we only provide the test name/error/line if the new generated patch has a different failure than the original (Section 3.2). This allows us to more concretely inform ChatGPT if it has made some progress in fixing a bug (e.g., patch no longer crashes with null-pointer exception but fails on some other test).

Initially, we see that the base response message achieves the worst result in number of bugs fixed. Similar to the behavior of the initial prompts, we can improve performance by adding the name of the failing test, error message along with the exact failure line from the failing test. Additionally, we can further improve performance by using the dynamic feedback response. Since in the initial prompt we already provide ChatGPT with the failing test name, error message, and failing line, in dynamic feedback response, we only provide new data if the generated patch contains a different failing information. This allow us to make more use of the conversational aspect by referring to a previous message. Furthermore, it can reduce the cost by using a short concise message if the patch does not make additional progress.

5.3.3 Conversation Length. Figure 7 shows performance in both the number of plausible fixes and the average dollar cost to fix a bug across different maximum conversation length. Recall from Section 3.2 that the maximum conversation length dictates the amount of history/feedback within each individual repair conversation, where length = 1 is equivalent to sampling using the initial prompt without any feedback. We observe that directly sampling from the ChatGPT without any conversation achieves the lowest number of plausible fixes. As we add the conversation element of ChatRepair, the number of plausible patches improves. Compared with sampling from the same prompt over and over again, by using ChatRepair in a conversational manner, the model can learn from its previous mistakes along with the concise test failure feedback information to generate more plausible patches. We also notice that the model can retain its performances as we increase the conversation length to be higher (i.e., 5 and 6). However, we see that compared with a lower conversation length (3), the higher conversation lengths incurs a much higher cost in fixing a bug. The reason is that as we increase the length, the amount of history/context (tokens) processed by the model will be higher, leading to higher cost per bug fixed. Our default conversation length of 3 serves as a good balance between cost and the number of bugs fixed.

5.4 RQ4: Evaluation on Recent Bugs

Table 6: Correct and Plausible fixes on ConDefect

Tools	ConDefects-Java	ConDefects-Python
ChatRepair	**243 / 250**	**241 / 249**
BaseChatGPT	170 / 189	165 / 171
AlphaRepair	154 / 158	142 / 160

To address potential data leakage of the groundtruth patches in prior evaluation datasets, we further evaluate on ConDefect [69], a collection of programming contest bugs obtained after the knowledge cut-off date of ChatGPT. Table 6 shows the repair performance (# of correct / # plausible patches) of ChatRepair, BaseChatGPT and AlphaRepair on ConDefect dataset in both Java and Python. We first observe that on both ConDefect datasets, ChatRepair is able to significantly outperform the repair baseline with 57.8% and 69.7% more correct patches on Java and Python respectively. Additionally, we see that ChatRepair also improves over the baseline technique of naively using ChatGPT (BaseChatGPT) by 42.9% and 46.6% more correct patches. ChatRepair successfully utilizes the previously ignored semantic information in test failures as well as prior failing and plausible fix attempts to achieve the state-of-the-art results on ConDefect.

6 Limitations & Future Work

First, while ChatRepair is able to achieve state-of-the-art repair performance of the studied benchmarks, there are still bugs that it struggles to fix. One particular category are multi-hunk bugs that require patches across multiple functions or files. This limitation is due to the limited context window of ChatGPT and other LLMs where they can only take in a restricted number of tokens as inputs. As such, LLM-based approaches like ChatRepair in its current iteration cannot deal with bugs that require edits to multiple different files. Also due to the context window size, ChatRepair does not have access to project and repository specific information. Prior work [70] has demonstrated the importance of learning and using project specific information (e.g., common variable, class, function names) in helping to boost LLM repair performance. ChatRepair currently can only obtain the context from the limited input and thus may miss the crucial intra-project information that is useful for repair.

To address these limitations, we aim to further improve ChatRepair with LLM-based agents. We plan to first reduce the reliance on traditional fault localization tools by using LLMs to identify potential buggy locations. LLMs, through its powerful code understanding ability, can identify further relationships between multiple buggy locations in order to aid in multi-hunk repair. Further, we can improve ChatRepair with a retrieval agent that retrieves and obtains useful repair-specific information. For example, by providing a summary of the related class fields or methods as part of the input to fix the current buggy class. We hope our agent-based debugging approach can address the prior limitations and inspire future work in using LLM for more holistic APR and debugging.

7 Threats to Validity

Internal. The first internal threat comes from the manual validation used to determine the correctness of the plausible patches compared with the reference developer patch. To address this, following prior work [29, 71, 77, 83], we carefully examined each patch and have released the correct patches along with the code [15].

Another threat to validity comes from the data leakage of reference developer patches being part of the original training data of ChatGPT. Since ChatGPT is a proprietary model and can only be accessed through API, we do not have access to the exact training data used. To address this, we follow prior work [72] and first compute the number of correct patches generated by ChatRepair which was the same as the reference developer patch on Defects4j 1.2. We found that out of 212 (adding all correct patches from three repair scenarios) correct patches, 77 of them is the same as reference developer fix (36%). In addition, even if we remove all correct patches (77) which are the same as the reference developer patch, ChatRepair is still able to generate the correct patch for 8 unique bugs that none of the prior approaches can fix. Also, compared to the base ChatGPT repair baseline which uses the same underlying model, ChatRepair is able to drastically improve its performance (34 more correct fixes) showing that the result gained by ChatRepair is not simply due to memorizing the training data. Furthermore, we evaluate ChatRepair on the ConDefect dataset collected after the knowledge cut-off date of ChatGPT and demonstrate that ChatRepair is able to improve over the best baseline with 42.9% and 46.6% more correct patches on the Java and Python version respectively. To completely address this threat, we would need to retrain ChatGPT from scratch which would be infeasible for an academic project.

Additional threat to validity is our evaluation setting of using perfect fault localization (PFL). We first note that PFL is the preferred evaluation setting for prior LLM-based and traditional APR tools [68, 70, 71] to reduce effect different fault localization tools have on repair. To address this threat, we conducted an initial study by using widely adopted fault localization (Ochiai [86]) and assumed that ChatRepair may only fix a bug if the groundtruth location is part of the top-40 suspicious locations. We reduce the number of patches to 20 per line, incurring a total time cost of <5 hour bug (typical setting for non-perfect fault localization). In this experiment, ChatRepair is still able to achieve 68 bugs fixed compared to state-of-the-art baseline [72] of 50 bugs fixed without groundtruth buggy location.
External. The main external threat to validity comes for our evaluation datasets used. The improvement obtained by ChatRepair may not generalize to other repair datasets. To address this, we evaluate on Defects4j 2.0, two QuixBugs datasets and the recently introduced ConDefect dataset to demonstrate the generalizability.

8 Conclusion

We propose ChatRepair – the first fully automated conversation-driven APR tool which leverages ChatGPT to perform repair. ChatRepair learns from both previously incorrect and plausible patches and utilizes test failure information to provide immediate and dynamic feedback for patch generation. Using our conversational repair paradigm, ChatRepair is able to achieve the new state-of-the-art performance of 114 and 48 bugs on Defects4j 1.2 and 2.0 respectively.

Acknowledgements

We thank reviewers across multiple conferences for their insightful feedback and comments to improve this paper. This work was partially supported by NSF grant CCF-2131943 and Kwai Inc.

References

[1] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[2] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *FSE 2014*. 306–317.

[3] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *ASE*. 907–918.

[4] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*. arXiv:2204.06745.

[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] chatgptendpoint 2023. Introducing ChatGPT and Whisper APIs. https://openai.com/blog/introducing-chatgpt-and-whisper-apis.

[8] chatgptguide 2023. ChatGPT Guide. https://platform.openai.com/docs/guides/chat.

[9] chatgptweb 2023. ChatGPT Web. https://chat.openai.com/chat.

[10] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *arXiv preprint arXiv:2406.01304* (2024).

[11] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[14] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transaction on Software Engineering* (2019).

[15] Dataset 2023. Dataset. https://figshare.com/s/9796028cef4d7dbc08ff.

[16] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. 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* (Hyderabad, India) *(CSTVA 2014)*. 30–39.

[17] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. In *46th International Conference on Software Engineering (ICSE)*.

[18] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 85–91.

[19] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. *arXiv preprint arXiv:2405.11514* (2024).

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155.

[21] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999.

[22] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[23] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, 19–30.

[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[25] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. 837–847.

[26] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, 888–891.

[27] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 298–309.

[28] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[29] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (May 2021).

[30] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting concise bug-fixing patches from human-written patches in version control systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 686–698.

[31] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440.

[32] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop*.

[33] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024.

[34] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[35] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224.

[36] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 3–13.

[37] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[38] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614.

[39] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 55–56.

[40] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 31–42.

[41] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. 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, 456–467.

[42] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[43] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. New York, NY, USA, 166–178.

[44] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. 702–713.

[45] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automatd program repair refine fault localization? A unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Los Angeles, California, United States, July 18-22 , 2020*. 12 pages.

[46] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114.

[47] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? *arXiv preprint arXiv:2406.01422* (2024).

[48] Matias Martinez, Thomas Durieux, Jifeng Xuan, Romain Sommerard, and Martin Monperrus. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. arXiv:1505.07002.

[49] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444.

[50] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. 691–701.

[51] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *ICSE 2023*. 1456–1468.

[52] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[54] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[55] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[56] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 69–75.

[57] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[58] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439.

[59] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[60] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347.

[61] John Schulman, Barret Zoph, Jacob Hilton Christina Kim, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, Rapha Gontijo Lopes, Shengjia Zhao, Arun Vijayvergiya, Eric Sigler, Adam Perelman, Chelsea Voss, Mike Heaton, Joel Parish, Dave Cummings, Rajeev Nayak, Valerie Balcom, David Schnurr, Tomer Kaftan, Chris Hallacy, Nicholas Turley, Noah Deutsch, Vik Goel, Jonathan Ward, Aris Konstantinidis, Wojciech Zaremba, Long Ouyang, Leonard Bogdonoff, Joshua Gross, David Medina, Sarah Yoo, Teddy Lee, Ryan Lowe, Dan Mossing, Joost Huizinga, Roger Jiang, Carroll Wainwright, Diogo Almeida, Steph Lin, Marvin Zhang, Kai Xiao, Katarina Slama, Steven Bills, Alex Gray, Jan Leike, Jakub Pachocki, Phil Tillet, Shantanu Jain, Greg Brockman, and Nick Ryder. 2022. ChatGPT: Optimizing Language Models for Dialogue. (2022). https://openai.com/blog/chatgpt/.

[62] Jessica Shieh. 2023. Best practices for prompt engineering with OpenAI API. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api.

[63] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. 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*. 532–543.

[64] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. arXiv:1409.3215.

[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. (2017). arXiv:1706.03762.

[66] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741* (2024).

[67] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. 1–11.

[68] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[69] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2023. Condefects: A new dataset to address the data leakage concern for llm-based fault localization and program repair. *arXiv preprint arXiv:2310.16253* (2023).

[70] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering (ICSE '23)*.

[72] Chunqiu Steven Xia and Lingming Zhang. 2022. 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 (ESEC/FSE 2022)*.

[73] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. In *Deep Learning for Code Workshop*.

[74] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[75] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563* (2023).

[76] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).

[77] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE22)*. Association for Computing Machinery, Article 92, 13 pages.

[78] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1506–1518.

[79] Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, et al. 2023. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. *arXiv preprint arXiv:2303.10420* (2023).

[80] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[81] Shafiq Joty Yue Wang, Weishi Wang and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[82] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE]

[83] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. 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*. ACM, New York, NY, USA, 341–353.

[84] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *ICSE 2023*.

[85] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2019. Fine-Tuning Language Models from Human Preferences. arXiv:1909.08593.

[86] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.

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