CONVERSATIONAL AUTOMATED PROGRAM REPAIR

Chunqiu Steven Xia, Lingming Zhang
University of Illinois at Urbana-Champaign
{chunqiu2, lingming}@illinois.edu

ABSTRACT

Automated Program Repair (APR) can help developers automatically generate patches for bugs. Due to the impressive performance obtained using Large Pre-Trained Language Models (LLMs) on many code related tasks, researchers have started to directly use LLMs for APR. However, prior approaches simply repeatedly sample the LLM given the same constructed input/prompt created from the original buggy code, which not only leads to generating the same incorrect patches repeatedly but also miss the critical information in testcases. To address these limitations, we propose *conversational APR*, a new paradigm for program repair that alternates between patch generation and validation in a conversational manner. In conversational APR, we iteratively build the input to the model by combining previously generated patches with validation feedback. As such, we leverage the long-term context window of LLMs to not only avoid generating previously incorrect patches but also incorporate validation feedback to help the model understand the semantic meaning of the program under test. We evaluate 10 different LLM including the newly developed ChatGPT model to demonstrate the improvement of conversational APR over the prior LLM for APR approach.

1 INTRODUCTION

Bugs in software can cause significant financial losses Matteson (2018) and create dangerous health and safety problems Hanbury (2019). Due to the high manual cost of fixing bugs O'Dell (2017), Automated Program Repair (APR) Gazzola et al. (2019) is a promising solution to reduce developer work by automatically generating patches given the buggy code and failing testcases.

Traditionally, APR approaches commonly use the paradigm of Generate and Validate (G&V), where APR tools will first generate a list of candidate patches given the original buggy code and then validate each one sequentially until a *plausible patch* that passes all the testcases is found. Plausible patch is then passed on to a human developer where they have to determine if this is a *correct patch* that correctly fixes the underlying bug. Traditional APR approaches such as template-based tools Ghanbari et al. (2019); Liu et al. (2019); Lou et al. (2020) have been proven useful in fixing bugs with pre-defined templates to match buggy and corresponding fix code patterns. Recently, researchers have designed learning-based APR tools Ye et al. (2022); Zhu et al. (2021); Jiang et al. (2021) which build a Neural Machine Translation (NMT) model by training on pairs of buggy and patch code. However, these learning-based APR tools suffer from lack of patch variety as it can only repair the types of bugs that are a part of the buggy/patch training data. Furthermore, these bug fixing datasets can be difficult to construct as it require scraping open-source bug fix commits which may contain many false positives, adding noise to the dataset.

Recognizing the limitation of prior learning-based APR tools, researchers have started to look into directly leveraging Large Pre-Trained Language Models (LLMs) for APR without fine-tuning. LLMs have proven their ability in various code generation tasks Austin et al. (2021). Xia & Zhang (2022) first introduced *cloze-style* APR where a LLM directly fill-in the correct code given its surrounding context. Other studies Prenner et al. (2022); Kolak et al. (2022); Xia et al. (2022) have also investigated directly applying different types of LLMs for APR by smartly applying prompts or giving original buggy code as context. Typically, directly applying LLMs for APR involves creating a common prompt/prefix which can be just the buggy context (zero-shot) or combining buggy context with a few examples of bug fixes (few-shot) as input to the model. Following the G&V paradigm, prior approach will sample the LLMs multiple times to obtain candidate patches. However, this pipeline has the following limitations:

First, sampling from the same prefix/prompt multiple times can lead to many repeated patches due to the probabilistic nature of sampling. This means the LLMs may waste a lot of compute and time generating the same patches which have already been validated as incorrect by the testsuite. Second, prompts provided to the LLMs for APR are created only from the original buggy code and does not include any of the testcase information. Such information like the expected input and output examples that can help LLMs understand the functionality of the buggy program are not provided. Third, prior approaches also fail to consider the outputs produced by the generated incorrect patches. Previously incorrect patches may fail on a particular corner case, which can be exposed by looking at the test output and providing it to the LLM to address it in future patches.

Our Work. We propose *conversational APR* – a new paradigm of using LLMs for APR that directly leverages the testcase validation information to provide feedback to LLMs in a conversational manner. In conversational APR, we interleave patch generation with validation where LLM first generates a patch, we then validate it against testsuite to provide feedback and prompt LLM with the new feedback information to generate a new patch. While in this paper we consider simple testcase input/output/error validation feedback, one can apply conversational APR with a wild range of possible feedback information such as human evaluation of the patch. We refer to the process of generating a patch followed by validation as a *turn* where a conversation *chain* is made up of multiple turns in sequence. In the start of the conversation chain, we begin with an initial prompt and sample the LLM to obtain a candidate patch. As we continue the conversation, the input given to the LLM in each turn is a concatenation of all previously incorrect patches along with their associated testcase feedback within the same conversation chain. A conversational chain is terminated once a patch that passes all the testcases are found or the maximum chain length is reached (i.e., maximum number of turns). In the latter case, we start a new conversation chain with the initial prompt again.

Compared with prior LLM for APR tools which only use the buggy code snippet as inputs, conversational APR incorporates patch validation in the form of validation feedback to help the model understand the *reason* why previously generated patches are incorrect. Such feedback can contain the incorrect and expected test outputs or indicate if the generated patch contains compilation/runtime errors. Furthermore, while prior LLM for APR tools continuously sample from the same input, our approach iteratively builds the input by including previously incorrect patches. As such, the LLM, through its long context window, can recognize previous generations and avoid repeatedly generating an already validated incorrect patch. We evaluated our conversational APR by using 10 popular LLMs, where we found that our approach not only improves the number of bugs fixed but also can arrive at the correct patch faster compared with sampling-based baseline. Furthermore, we also evaluate the recently developed ChatGPT Schulman et al. (2022)[1], a dialogue focused LLM trained using reinforcement learning and highlight the performance of conversational APR when using a LLM designed for conversation/dialogue.

2 Background & Related Work

2.1 LLMs for APR

To combat the reliance on training using bug-fixing datasets to build learning-based APR tools based on NMT models, researchers directly applied LLMs for APR without any fine-tuning. Xia & Zhang (2022) proposed AlphaRepair, the first *cloze-style* APR to directly leverage LLMs for APR in a zero-shot setting by removing the buggy line and replacing it with masked tokens. AlphaRepair then queries the CodeBERT Feng et al. (2020) model to fill-in the masked tokens with the correct tokens to generate patches. Prenner et al. (2022) investigated the ability for Codex Chen et al. (2021) to repair bugs using a simple prompting method to generate a complete patched function given the original buggy function. Kolak et al. (2022) evaluated the scaling effect of LLMs for APR by using 4 LLMs of different model sizes to generate a single line fix given only the original buggy prefix (i.e., removing all lines after and including the buggy line of the buggy function). Recently, Xia et al. (2022) conducted an extensive study on directly applying LLMs for APR. In the study, they adopt

[1] While we perform repair using ChatGPT, no part of this paper is written by ChatGPT. :)

several repair settings, including few-shot generation using a few examples of bug fixes, cloze-style APR and also single line generation.

The findings across these prior work is consistent in showing that directly using LLMs for APR achieves comparable if not better performance compared to prior APR tools. However, these proposed LLMs for APR techniques almost exclusively use sampling where patches are generated by sampling from the same input over and over again, leading to many repeated patches. Furthermore, the inputs to the LLMs are only constructed from the original buggy function, missing the rich information in the form of testcases. In this work, our conversational APR approach aims to bridge these limitations in LLMs for APR by constructing new inputs based on prior incorrect patches to avoid sampling repeated patches and providing the validation feedback to add another dimension of input apart from original buggy code to help the model understand the semantic meaning of the program.

2.2 Multi-Step Program Reasoning and Synthesis using LLMs

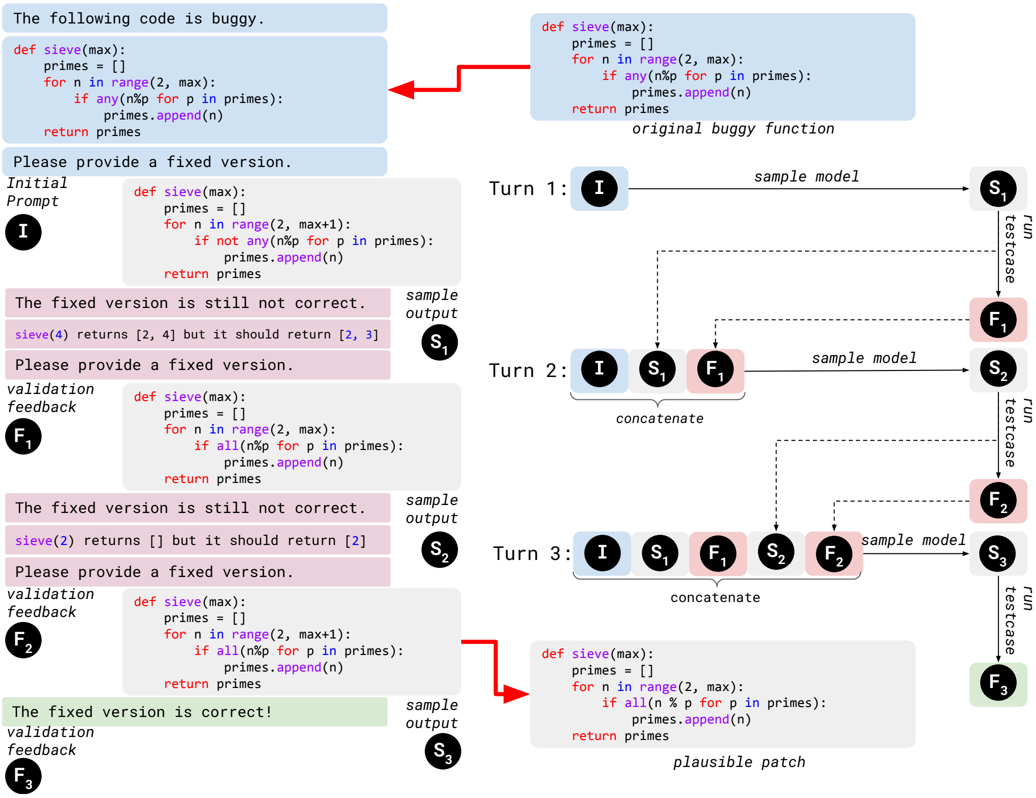
A related research direction is in applying multi-step reasoning for code understanding and synthesis. Nye et al. (2021) trains a LLM designed for program understanding by introducing the idea of a "scratchpad" in which the LLM predicts the intermediate states of a program along with the final execution results. Chen et al. (2022) extends the chain-of-thoughts Wei et al. (2022) prompting style in NLP to propose program-of-thoughts where the prompt contains an explicit command to construct the program step-by-step. However, these work still generates a complete result (i.e., final program execution or code), albeit with intermediate results, in one shot, whereas our conversational APR samples multiple times LLMs with different inputs to obtain one output plausible patch.

Different from one-shot methods, Austin et al. (2021) investigated the ability for LLMs to use human feedback in a conversational manner for program synthesis. The approach works by keeping a conversation of previously generated code and correcting any mistake using natural language feedback provided by human developers. Nijkamp et al. (2022) manually created a multi-step synthesis dataset where each target program is broken down into multiple smaller steps where only a few lines of code needs to be generated. They then sample the model multiple times to iteratively complete each smaller step and concatenate them together to form the final program. While these described techniques involve iteratively sampling from the model with new feedback similar to a conversational manner, our work can automatically create this feedback through testcase execution without any human-in-the-loop.

3 Conversational APR

We propose a conversational APR approach to prompt LLM patch generation by combining previously generated patches and validation feedback in a conversational manner. Contrasting with the classic Generate and Validate (G&V) APR approach that first generates a large number of candidate patches and then validate each one to find a list of plausible patches, conversational APR interleaves generation and validation to provide immediate feedback for the new candidate patch. Different from previous APR tools which make use of LLMs through sampling given the same prefix/context for each bug, conversational APR approach aims to incorporate feedback information after each generation (if the candidate patch failed to pass all tests) as new context for subsequent generations. Specifically, the feedback information includes both the incorrect generated patch and its associated failed testcase information.

Conversational APR involves iteratively obtaining new candidate patches from the LLM by using previously generated patches/validation results as feedback. We refer to this process as a *turn*, where each turn includes three different steps: 1) construct new a prompt based on prior feedback 2) sample the model to produce a sample output function 3) validate the sample output function against testcases to obtain validation feedback. Multiple turns in sequence is defined as a *chain*. The terminating conditions are that the sample output patch is able to pass all testcases (i.e., a plausible patch is obtained) or the maximum number of turns (length of the chain) is reached. Note that each turn (all three steps) are done automatically without needing any human-in-the-loop, this allows conversational APR to be an automatic approach for program repair.



The following code is buggy.

```
def sieve(max):
    primes = []
    for n in range(2, max):
        if any(n%p for p in primes):
            primes.append(n)
    return primes
```

Please provide a fixed version.

```
def sieve(max):
    primes = []
    for n in range(2, max+1):
        if not any(n%p for p in primes):
            primes.append(n)
    return primes
```

The fixed version is still not correct.

sieve(4) returns [2, 4] but it should return [2, 3]

Please provide a fixed version.

```
def sieve(max):
    primes = []
    for n in range(2, max):
        if all(n%p for p in primes):
            primes.append(n)
    return primes
```

The fixed version is still not correct.

sieve(2) returns [] but it should return [2]

Please provide a fixed version.

```
def sieve(max):
    primes = []
    for n in range(2, max+1):
        if all(n%p for p in primes):
            primes.append(n)
    return primes
```

The fixed version is correct!

```
def sieve(max):
    primes = []
    for n in range(2, max):
        if any(n%p for p in primes):
            primes.append(n)
    return primes
```

original buggy function

```
def sieve(max):
    primes = []
    for n in range(2, max):
        if all(n % p for p in primes):
            primes.append(n)
    return primes
```

plausible patch

Figure 1: Overview of conversational APR with an illustrative example in fixing the buggy `sieve` function

3.1 PIPELINE & EXAMPLE

Figure 1 shows an illustrative example of a conversation chain (multiple turns) and an overview of the pipeline of the conversational APR approach. We first take in as input the original buggy function and a set of testcases which contains some failing tests that expose the underlying bug. In the example, the buggy function (`sieve`) attempts to use to sieve algorithm to calculate the list of prime numbers below the integer input (`max`). The location of the bug occurs on line 4 where the buggy function incorrectly uses `any` instead of `all`. This bug is exposed by the testcase of `sieve(2) = [2]` where the buggy function incorrectly returns an empty array `[]`.

- **Turn 1:** We first create an initial prompt I using the original buggy function which contains natural language description to indicate that the function is buggy (`The following code is buggy`) and the task we want the LLM to solve (`Please provide a fixed version`). We then sample the model using the initial prompt I to obtain the first sample output function S_1. The change is made to line 4 where the function in S_1 negated the original `if` condition. We then validate S_1 against the list of tests and found that while the new patch is able to successfully pass the previous failing test of `sieve(2) = [2]`, it returns `[2, 4]` for `sieve(4)` when the correct output should be `[2, 3]`. This validation information F_1 is collected as feedback to use during the next conversation turn.
- **Turn 2:** Different from turn 1, where the input to the LLM is just the initial prompt I, now we provide the model also with the previously generated patch and its failing testcase. In short, we construct the validation feedback F_1 by using the failing testcase and indicate to the model that the previous sample S_1 is still not correct (`The fixed version is still not correct`) and the new task (`Please provide another fixed version`). We then concatenate the initial prompt, first sample output function and the validation feedback {I, S_1, F_1} together as the input to the LLM. As such, the model is able to not only use the original buggy function but also use the previously generated sample and its testcase feedback to generate a new patched function. Similar to turn 1, we obtain S_2 and F_2 where the correct line 4 is obtained (switching `any` to `all`) but the candidate patch function incorrectly reduced the upper range of the for loop by 1.

- **Turn 3:** Similar to turn 2, we first construct the new validation feedback F_2 from the previous failing test case. We then concatenate all previously sampled output along with its validation feedback in sequence to produce {I, S_1, F_1, S_2, F_2}. Using this input, we then sample the LLM again to produce the next candidate patch S_3. We observe that this candidate patch correctly fixes the underlying bug and this is indicated by its validation F_3 where it is able to pass all the testcases. The program repair process is then terminated as we have obtained our plausible patch S_3.

Compared to prior approach in APR based on LLMs which simply samples from a pre-defined prompt/context, conversational APR leverages the previously missing key feedback information in the form of testcase results to prompt future patch generations. The testcase feedback not only tells the LLM that the previous patches are incorrect (i.e. leading to more unique patches) but also provides input and output examples which helps the model to understand the underlying functionality of the function (i.e. leading to more correct patches).

3.2 Design Decisions

In the above example illustrated in Figure 1, we show the overall pipeline of conversational APR. However, there are different design decisions which can impact the performance of the approach:

Prompt engineering. Prompting has been shown to be an effective way of leveraging LLMs on various downstream tasks without needing any explicit fine-tuning. In conversational APR approach, we follow the style of prior work Xia et al. (2022) in providing a short and concise prompt with respect to the description of the input and the task we want to model to solve. Additionally, we follow prior guidelines and kept the prompt to be open-ended rather than to restrict the generation with a close-ended prompt. One particular important prompt constructing is validation feedback in providing the failing testcase to the LLM. In the Figure 1 example, we provide a *functional* prompt that directly invokes the function and highlight the discrepancy between output and expected testcase output. We refer to this as functional prompt since it directly calls the function with input parameters similar to what one would do in code. In Section 6.2, we compare this style of validation prompting with other methods including without any testcase information to demonstrate the benefit of including validation feedback to the model.

Maximum chain length. Recall that a conversation chain refers to the continuous sequence of turns to fix a bug. A chain is demonstrated in Figure 1 with a chain length of 3. Along with finding a plausible patch, a preset value for the maximum chain length is also a terminating condition since the LLM used will have a maximum context window and cannot take in arbitrary length inputs. Once this maximum chain length is reached, conversational APR will restart from the beginning (i.e., by crafting initial prompt again) with a new chain conversation. The maximum chain length is a parameter which controls how much *history* the LLM may receive. A maximum chain length of 1 refers to the base case of sampling from the initial prompt over and over again, meaning the model does not know any of the previously generated incorrect patches. A higher maximum chain length means the model can see multiple previously failed patches, however this also may not be beneficial as it can cause the LLM to repeat some of the earlier patches or get stuck on a particular implementation of the function. In Section 6.2, we evaluate the effect of the chain length has on repair performance.

4 Datasets

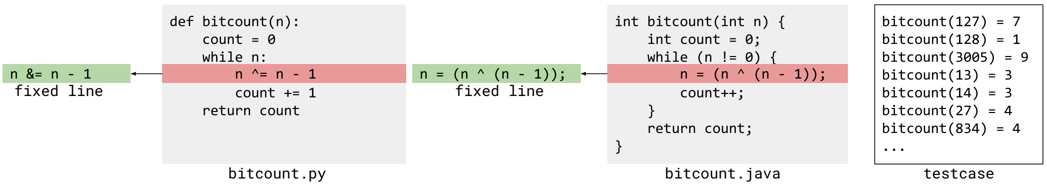
In this section, we describe the LLMs used in our evaluation and also the repair benchmark used to evaluate our proposed technique.

4.1 LLMs

In our work, we evaluate 10 different LLMs to not only demonstrate the effect of scaling behavior on our proposed conversational APR approach but also to evaluate how different pre-training and model design contribute to the overall effectiveness. Table 1 presents an overview of the studied LLMs. Column **Model** is the model name, **#Parameters** indicates the number of model parameters, **Context Window** represents the size of the context window, and **Training Strategy** refers to the training strategy used.

Table 1: Evaluation LLM overview

Model	#Parameters	Context Window	Training Strategy
CodeGen-Mono	350M/2B/6B/16B	2048	Unsupervised CLM
CodeGen-Multi	350M/2B/6B/16B	2048	Unsupervised CLM
Codex	12B	4096	Unsupervised CLM
ChatGPT	$\sim$175B	$\sim$4000	Reinforcement Learning from Human Feedback + CLM



```
def bitcount(n):
    count = 0
    while n:
        n ^= n - 1
        count += 1
    return count
```

```
n &= n - 1
```

```
int bitcount(int n) {
    int count = 0;
    while (n != 0) {
        n = (n ^ (n - 1));
        count++;
    }
    return count;
}
```

```
n = (n ^ (n - 1));
```

```
bitcount(127) = 7
bitcount(128) = 1
bitcount(3005) = 9
bitcount(13) = 3
bitcount(14) = 3
bitcount(27) = 4
bitcount(834) = 4
...
```

Figure 2: Example bug in both Python and Java in QuixBugs along with the testcases

- **CodeGen Nijkamp et al. (2022).** A family of autoregressive LLMs trained using Causal Language Modeling (CLM) objective (next-token-prediction) ranging from 350M to 16B in parameter size. CodeGen is first trained on the open-source ThePile Gao et al. (2020), containing 22 diverse text-based datasets. The models are then trained on BigQuery BigQuery, a dataset of open-source code from 6 programming languages. We refer to these models (trained on ThePile then BigQuery) as CodeGen-Multi. CodeGen-Multi is then further trained on a dataset containing large amounts of Python GitHub code to produce CodeGen-Mono. In our experiments, we use CodeGen-Mono for repair benchmarks in Python and CodeGen-Multi for repair benchmarks in other programming languages by refer to them both as CodeGen for simplicity.
- **Codex Chen et al. (2021).** A programming language focused autoregressive model based on the GPT-3 architecture Brown et al. (2020). Codex is first initialized with GPT-3 weights from training on natural language corpus and then fine-tuned using next-token-prediction on a large dataset of code files. While Codex also contains a version which can take in suffix tokens (i.e., fill-in code in the middle), for our experiments, we only use Codex by providing the prefix context.
- **ChatGPT Schulman et al. (2022).** A conversational-based LLM first initialized from GPT-3.5 model and then fine-tuned using Reinforcement Learning from Human Feedback (RLHF) Ziegler et al. (2019). ChatGPT is first fine-tuned based on supervised learning where human provides example responses to prompts in the dataset. Using this fine-tuned model, a reward model is then trained by sampling multiple outputs of the model from a given prompt and again using a human to rank the outputs. The reward model is used in the reinforcement learning step where Proximal Policy Optimization Schulman et al. (2017) is used to fine-tune ChatGPT. Different from the Codex and CodeGen, ChatGPT through the usage of RLHF and fine-tuning data is designed for conversation where the usage encourages a dialogue format. Note that much of the ChatGPT model detail is unknown to the public, therefore, we can only provide an approximate value for the number of parameters[2] and context window size OpenAI (2022) according to verified sources.

4.2 Benchmarks

We use the QuixBugs Lin et al. (2017) repair benchmark to evaluate our proposed conversational APR approach. QuixBugs has been widely used to evaluate many repair tools including both learning-based Ye et al. (2022); Zhu et al. (2021); Jiang et al. (2021); Drain et al. (2021) and LLM for APR Xia & Zhang (2022); Xia et al. (2022); Kolak et al. (2022); Prenner et al. (2022) approaches. QuixBugs dataset contains the same 40 bugs and it associated correct patch in both Python and Java. These bugs are self contained functions based on classic algorithms and it usually only takes a single line change to fix the underlying bug. Each bug comes with a set of testcases which the buggy function failed to pass and can be used to evaluate any candidate patch generated. Figure 2 shows an example bug for the `bitcount` function in QuixBugs for both Java and Python. The bug occurs inside the while loop where the code incorrectly uses the ^ operator instead of `&` operator. We also show the example testcases for `bitcount` where it contains example inputs and the expected outputs when evaluated using the function.

[2] As ChatGPT is fine-tuned on GPT-3.5, we assume a similar number of parameters as GPT-3.5

Out of the 40 bugs in QuixBugs, we further filter out 10 bugs which includes testcases that are difficult to represent with our validation feedback prompt. For example, testcases for `detect_cycle` involves a graph as an input to the function. In total, we use 60 bugs (30 and 30 respectively for Java and Python) for our evaluation.

5 Experimental Setup

In this section, we describe the key research questions that our evaluation seek to answer, the evaluation metrics used and also describe the implementation details.

5.1 Research Questions

We aim to investigate the following research questions:

- **RQ1:** What is the effectiveness of applying conversational APR?
- **RQ2:** How do different components of conversational APR effect performance?

In RQ1, we first compare the performance of conversational APR with a baseline approach used in prior LLM for APR work where the patches are generated by continuously sampling from the same initial prompt. We further evaluate both the scaling effective of LLM as we increase the size of the model and also investigate the difference in performance of different pre-training strategies (e.g., ChatGPT vs. Codex). In RQ2, we dive deeper into the different parameters of conversational APR. Specifically, we evaluate how the length of the conversational chain and different validation feedback prompts affect the performance.

5.2 Evaluation Metrics

Our evaluation metric consist of the standard metric used to evaluate APR tools: number of *plausible patches*: patches which passes all the testcases and *correct patches*: patches which are semantically equivalent to the reference developer patch. Additionally, since we are using sampling LLMs, we also define *tries* as the number of samples needed to obtain a plausible/correct patch. This metric is useful when comparing two approaches/models that achieve similar number of bugs fixed, the one with fewer number of tries is preferred as we want to limit the number of times we have to sample the LLM.

5.3 Implementation

We implemented the LLM generation pipeline in Python using Hugging Face HuggingFace implementation of the CodeGen models. We access Codex through the OpenAI API by querying the *code-davinci-002* engine. Since ChatGPT is not open-sourced and does not provide an official API endpoint (like Codex), we manually input the prompt and extract the outputs. For all models apart from ChatGPT, we use a default generation setting of nucleus sampling with top p = 0.95, temperature = 1, 50 samples per bug with a maximum chain length of 3. We generate and evaluate patches on a 32-Core workstation with AMD Ryzen Threadripper PRO 5975WX CPU, 256 GB RAM and 3 NVIDIA GeForce RTX 3090 GPUs, running Ubuntu 22.04.1 LTS.

6 Results

6.1 RQ1: Conversational APR effectiveness

We first evaluate the effectiveness of applying conversational APR using validation feedback compared to prior method of sampling given the same prompt without any feedback. Table 2 shows the results on QuixBugs-Python and QuixBugs-Java. We observe that by *applying our feedback driven conversational APR, we are able to improve the # of correct and plausible patches for all unsupervisedly trained LLM across all model sizes*. Additionally, conversational APR is also able to decrease the # of tries (# of samples) needed before obtaining the first plausible/correct patch. Compared to traditional sampling method of producing patches, conversational APR is able to leverage the

Table 2: **Conversational APR performance on both QuixBugs-Python and QuixBugs-Java compared with baseline sampling method. #c/#p refers to the number of correct / plausible patches.**

Models	QuixBugs-Python				QuixBugs-Java			
	Sampling		Conversational		Sampling		Conversational	
	#c/#p	#tries	#c/#p	#tries	#c/#p	#tries	#c/#p	#tries
CodeGen-350M	7 / 10	20.5	8 / 11	18.4	4 / 4	24.2	5 / 5	23.5
CodeGen-2B	22 / 23	16.6	25 / 26	14.3	12 / 14	18.8	15 / 16	16.4
CodeGen-6B	22 / 24	14.0	27 / 28	12.1	18 / 20	19.8	22 / 22	13.5
CodeGen-16B	29 / 29	5.6	30 / 30	4.8	24 / 25	14.5	28 / 29	13.2
Codex	29 / 30	4.6	30 / 30	3.8	28 / 30	7.2	29 / 30	5.7

Table 3: ChatGPT and Codex comparison on QuixBugs-Python and QuixBugs-Java where each cell indicates the number of correct / plausible patches

Models	QuixBugs-Python			QuixBugs-Java		
	one try	two tries	three tries	one try	two tries	three tries
Codex	16 / 16	21 / 21	24 / 24	11 / 12	18 / 19	21 / 22
ChatGPT	24 / 24	27 / 28	28 / 29	24 / 24	26 / 26	26 / 26

model's understanding of natural language feedback to indicate why the patch is incorrect. LLMs can use this validation feedback information to generate new patches that try to pass the previously failed testcase. Furthermore, conversational APR also helps to reduce the number of repeated patches from sampling using the same prompt over and over again. By using the large context size of many state-of-the-art LLMs, conversational APR can use recently generated incorrect patches as previous context to prompt the model to generate a new patch that is different.

ChatGPT evaluation. We now evaluate the performance of ChatGPT when using conversational APR. Due to the requirement of manually inputting and extracting outputs from ChatGPT, we only use a single conversation chain with at most 3 tries (i.e. maximum chain length of 3). We compare with the best performing LLM of Codex from previous results under the same setting in Table 3. We observe that *compared to Codex, which is trained in an unsupervised manner, ChatGPT which is fine-tuned using Reinforcement Learning from Human Feedback (RLHF) performed much better across the two repair datasets*. This improvement in result can be partially attributed to increase in model parameter size, but we believe this is also due to the dialogue-based fine-tuning dataset used in ChatGPT. Conversational APR relies on the model understanding the validation feedback to condition the future generation in trying to generate a patch that passes the testcase. A more dialogue-oriented model such as ChatGPT is well suited for this task as both the training data and algorithm contain feedback driven loops. As ChatGPT and other dialogue-based LLMs become more popular, we believe conversational APR can also be further improved through more usage of these LLMs.

6.2 RQ2: Component Analysis

Maximum chain length. We first investigate the effect of different maximum chain length has on the repair performance. Figure 3 shows the number of plausible patches when we vary the maximum chain length from 1 to 6 for the 4 CodeGen models. Recall from Section 3 that chain length refers

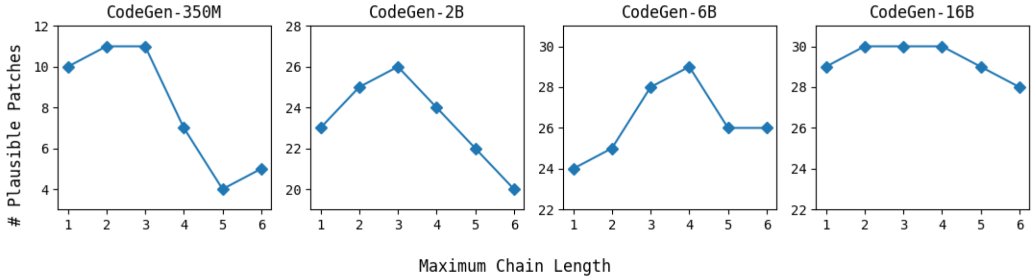


Figure 3: Number of plausible patches for the 4 different CodeGen models as we vary the maximum chain length on QuixBugs-Python

Table 4: Prompting Style Evaluation on QuixBugs-Python with each cell showing the number of plausible patches

Models	no testcase	natural language	functional
CODEGEN-350M	9	11	11
CODEGEN-2B	20	25	26
CODEGEN-6B	24	27	28
CODEGEN-16B	27	30	30
Codex	29	30	30

to the number of turns (each turn consist of generating and validating a new patch) in a conversation chain. A maximum chain length of 1 is the simple sampling from the same initial prompt baseline (used in prior LLM for APR tools). As we increase chain length, the model has to take in more and more previous context in the form of prior generations and feedbacks. We observe that the performance increase as we start from a small chain length and reaches the maximum around 3 or 4 and then decrease as chain length continue to increase. The decrease in number of plausible patches once we reach a high chain length is because the context may be too much for the model to handle since it can include multiple previously failed patches. We also observe that this decrease is more significant in smaller models, where larger models are less affected by longer chain length, showing the ability for larger models to better capture the long term context dependencies. This shows that the optimal chain length to use for conversational APR can be dependent on the individual LLM used.

Feedback prompting style. We now evaluate the effect of the feedback prompting style used in our conversational APR. Table 4 shows the number of plausible patches using different validation prompts in QuixBugs-Python. Column **no testcase** does not include any testcase feedback (only states that the patch is not correct), **natural language** describes the failing testcase (e.g., `when input is 2, the patch incorrectly returns [] but it should return [2]`) and **functional** which is the default prompting style discussed in Section 3. We observe that different prompting style does have an effect on the final performance of conversational APR. Starting from no testcase prompt, we can improve performance by adding specific testcase feedback information on top of telling the LLM that the patch is not correct. We also observe that the functional prompting style, using the buggy/patch function name and passing parameters (see Figure 1), performs the best. Functional prompting style conveys the failing testcase information in a more concise and natural way by phrasing the testcase input and expected output relationship as a function call.

7 CONCLUSION

We propose conversational APR, a new paradigm for program repair that interleaves patch generation with validation to provide immediate feedback for LLMs to better prompt future generated patches. Compared to previous LLM for APR approaches that only sample from the same input, conversational APR iteratively builds the input by concatenating previously incorrect patches and validation feedback. This allows for the model to avoid generating previously incorrect patches and also understand the semantic meaning of the function through validation feedback. Our evaluation on 10 different LLMs shows the improvement of conversational APR over the baseline sampling method used in prior LLM for APR tools. Furthermore, we demonstrate the promising future of applying ChatGPT, a conversational/dialogue driven LLM, for conversational APR, or APR in general for the first time.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. arXiv:2108.07732.

BigQuery. Bigquery github repos, 2022. `https://console.cloud.google.com/marketplace/details/github/github-repos`.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. arXiv:2005.14165.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. arXiv:2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2022. arXiv:2211.12588.

Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons, 2021. arXiv:2105.09352.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. arXiv:2002.08155.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. 2020. arXiv:2101.00027.

Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pp. 19–30. ACM, 2019. ISBN 978-1-4503-6224-5.

Mary Hanbury. Investigators have reportedly found more evidence that could connect the ethiopian boeing 737 max crash to a deadly accident five months before. *Business Insider*, 2019. `https://www.businessinsider.com/potential-link-between-ethiopian-boeing-737-max-crash-lion-air-mishap-2019-3`.

HuggingFace. Hugging face, 2022. `https://huggingface.co`.

Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021.

Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*, 2022.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multilingual program repair benchmark set based on the quixey challenge. SPLASH Companion 2017, pp. 55–56, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355148.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pp. 31–42, New York, NY, USA, 2019. ACM. ISBN 9781450362245.

Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 75–87, 2020.

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

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

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. arXiv:2112.00114.

Devon H. O'Dell. The debugging mindset. *acmqueue*, 2017. https://queue.acm.org/detail.cfm?id=3068754/.

OpenAI. Does chatgpt remember what happened earlier in the conversation? 2022. https://help.openai.com/en/articles/6787051-does-chatgpt-remember-what-happened-earlier-in-the-conversation/.

Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can openai's codex fix bugs?: An evaluation on quixbugs. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pp. 69–75, 2022.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. arXiv:1707.06347.

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. Chatgpt: Optimizing language models for dialogue. 2022. https://openai.com/blog/chatgpt/.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2022. arXiv:2201.11903.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning, 2022. arXiv:2207.08281.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models, 2022. arXiv:2210.14179.

He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1506–1518, 2022.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 341–353, New York, NY, USA, 2021. ACM. ISBN 9781450385626.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences, 2019. arXiv:1909.08593.