Universiteit Utrecht

The Power of Large Language Models: using OpenAI's ChatGPT for Automatic Patch Generation

Author:

Sylvain MAISSAN, 6007635

First Supervisor:

Johan T. JEURING

Second Supervisor:

Hieke W. KEUNING

August 3, 2023

Abstract

Automated Program Repair (APR) has emerged as a valuable tool for developers in the software development and maintenance process. Despite recent advances in deep learning (DL), the DL-based APR approaches still have limitations. A notable research gap exists in the current state-of-the-art (APR) methods, as they often require domain-specific knowledge and retraining when transitioning to different programming languages.

This study explores the potential of Large Language Models (LLMs), specifically ChatGPT, as a promising alternative for patch generation, as they can potentially overcome these limitations by not requiring domain knowledge and enabling seamless adaptation across different programming languages. The experiment focuses on exploring the potential of ChatGPT as a method for generating software patches. Specifically, we investigate its performance using the benchmark Defects4j v2.0, conducting tests on a total of 476 bugs. We assume perfect localization of the buggy lines for the purpose of the experiment.

In our analysis, we compare the results of the ChatGPT-based patch generation with other state-of-the-art APR methods. Our findings reveal that ChatGPT demonstrates a comparatively weaker performance in this context. However, despite its current limitations, our study highlights untapped potential within ChatGPT and other Large Language Models (LLMs). With ongoing advancements and improvements, it is plausible that LLMs may surpass existing methods and offer superior performance in the future. However, LLMs like ChatGPT need further improvements and refinements to fully realize their potential.

Contents

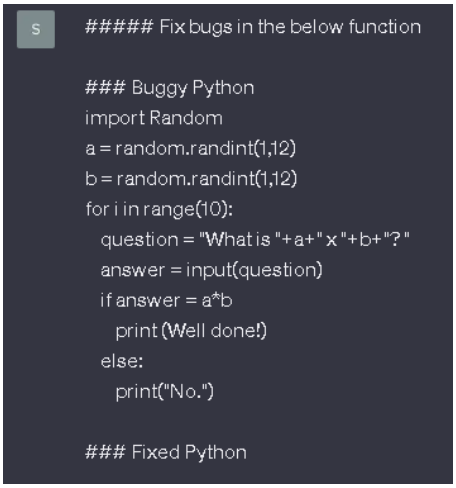
1 Introduction

Software systems surround us in daily life. They assist us and provide services that aim to improve our lives—for example, digital banking, healthcare diagnostic tools, and logistical route-planning systems. However, the consequence of the essential roles software systems play is that software bugs can be disastrous. For example, a bug in a rail company's operating system can halt all train traffic in the country.

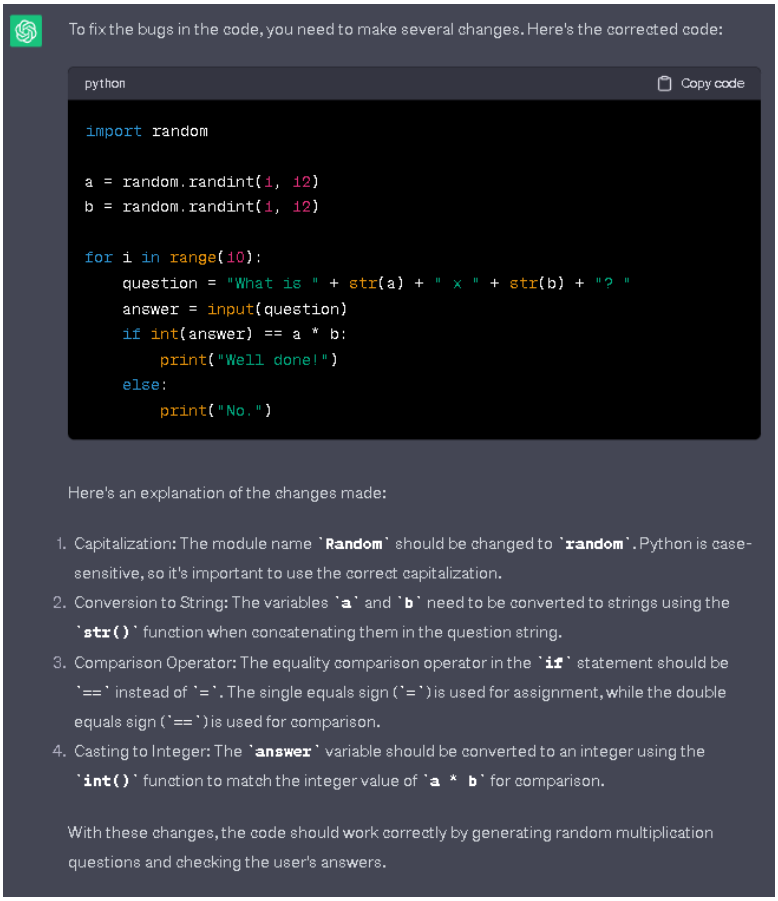
Moreover, bugs are ubiquitous, and finding and repairing them is a complicated, time-consuming, error-prone, and a manual process. A report shows that debugging and validating account for 35-50% of a software developer's time [1]. The cost of debugging, testing, and verification is estimated to account for 50-75% of a software product's total budget [2]. The excessively high cost of bug repair, plus the ever-growing importance because of the evolving software, motivated the research community to find a solution to this problem.

The proposed solution is to repair programs automatically instead of manually. This approach gives programmers more time to develop because the algorithm will fix the mistakes or oversights. The developer has more time to concern themselves with functionality requirements instead of implementation. This approach is called Automatic (or Automated) Program Repair (APR). These techniques reduce the cost of bug fixing by automatically generating patches for the developer. The aim is to generate fixes that a human developer would produce, i.e. to eliminate bugs without causing software regression and maintain readability of the code. Software regression is a phenomenon where a patch causes functionality loss. So, it might fix the original bug, but it also causes new bugs.

Recently OpenAI introduced a new GPT language model ChatGPT[3]. It is an GPT-3.5 model, which users can interact with in conversational way. The model has been a huge success with the public. The user-friendly chatbot interface has made the GPT model more



(a) The request



(b) The generated fix

Figure 1: Example of fixing a bug using ChatGPT.

approachable for people outside of the field of AI. However, its capabilities of generating texts has also raised a lot of questions about ethics, legislation, and labour. The model also possess the capabilities of one of its predecessor Codex. Codex is the GPT model that powers the newly released GitHub Copilot. Copilot is an AI pair-programming tool that assists the developer in the process of writing code. Codex, and therefore also ChatGPT, can understand and generate code up to a certain complexity. The models can execute various tasks (to a varying degree), such as generating code from natural language, explaining code, translating code between programming languages, providing time complexity, and bug fix-

ing. ChatGPT outperforms Codex on all of these tasks. In figure 1, there is an example of a prompt for bug repair using ChatGPT.

Moreover, The reason we believe that this experiment is interesting is that Large Language Models (LLMs) are great at zero-shot or few-shot learning[4]. Traditional APR techniques required substantial engineering efforts to develop the technique. Time spend in analysing the repair program, repairing faults, and custom tailoring to the domain language. On the other hand, ChatGPT is a general purpose model, which allows the model to adapt easily to the prompt given by the developer in any language. Furthermore, using a LLM removes the need for custom symbolic repair logic or retraining of a new model, and it allows us to handle both syntactic and semantic bugs [5].

In this paper, we investigate whether ChatGPT can be applied to the challenging task of APR patch generation. We evaluate and compare ChatGPT with other traditional APR techniques. For the comparison, we used a benchmark called Defects4j v2.0, a collection of reproducible bugs from open-source projects in Java. Additionally, we experiment which prompt yields the best performance. The prompt engineering is an important step for the results' accuracy. You could see prompt engineering as a part of hyperparameter tuning in traditional machine learning. The remainder of this proposal is structured as follows. Section 2 discusses experiments which are related to our reasearch. Section 3 provides background literature on APR, the learning-based patch generation, and GPT-3 language models. Section 4 presents the research methodology. In section 5, we will present the results and we will discuss the results more in depth in section 6. Finally, we provide our conclusion of the experiment in section 7.

2 Related Work

In this section, we will discuss other papers that utilize Large Language Models (LLMS) to tackle the APR problem. LLMs, which are typically employed for various Natural Language Processing (NLP) tasks, have also shown promise in programming languages by fine-tuning the model on code repositories, as exemplified by Codex[6]. A significant advantage of LLMs is their unsupervised learning approach, enabling them to be applied across different domains without requiring retraining. These models are trained on massive datasets containing billions of text/code tokens, and their architecture, based on transformers, excels at handling sequences, making them particularly effective in the NLP domain.

Prior research has already compared different LLMs on real-world projects [7], where nine LLMs were tested in different experimental settings: 1) complete function generation - aiming to generate the patched function from a buggy one, 2) correct code infilling - generating the correct replacement code given the prefix and suffix of the buggy function, and 3) single line generation - fixing a bug by making a single line change with the bug location provided. The experiments employed popular benchmark datasets, though multi-hunk bugs were excluded from the test set. A hunk refers to portion of the code that could potentially contain an error or bug. In other words, multiple hunks means that in order to fix the bug, you have to patch multiple locations in the source code. Furthermore, the paper does not specify how many patches are generated per bug. So, it is hard to compare the results because the unknown amount of resources. Next, the time cost is considerable for Codex, since it needs to be accessed through the OpenAI API, which also has a rate limit per account. Additionally, the tested LLMs failed to leverage the failing test as crucial information for bug fixing, and the absence of patch tracking led to computational inefficiency. The models produced identical solutions, which is a waste of computational resources.

Another study utilized ChatGPT in a conversational manner for patch generation [8, 9]. The approach adopted a conversational process by providing the ChatGPT with feedback and so iteratively refining the solution. The feedback sent back to the model was the error message from the console. This approach addresses the issue of repeated patches by leveraging previous answers. This allowed ChatGPT to identify plausible patches. Furthermore, whenever a plausible patch was found, the next step was to ask ChatGPT to generate alternative variations of this patch and produce additional candidate patches. This is more efficient way of procuring plausible patches, which is essential. Plausible patches have been shown to be valuable since they often share similar locations with the actual correct patches[10, 11]. Nevertheless, similar to previous work, multi-hunk bugs were not considered in the test set. Moreover, the conversational approach was slower compared to generating the response only once. Especially, since the intermediate response has to be tested if they pass the the test suite. Thus, the bug fix attempts are higher in quality, but cost more resources.

Moreover, there exists a study that combines traditional APR techniques with LLMs [12]. The method involves utilizing Codex to generate code for a given programming task, followed by employing APR tools like Tbar [13] and Recoder [14]. Despite this approach's potential benefits, it still inherits certain weaknesses inherent in traditional APR tools, such as a limited search space and a lack of awareness of program dependencies.

An intriguing observation is that LLMs tend to make common mistakes similar to those made by human programmers. This could be attributed to their training data, which largely consists of code written by humans. This insight underscores the significance of understanding the relationship between LLMs and human programming practices, as it opens up possibilities for more effective automated code repair techniques. Especially, since the goal is to write patches that are comprehensible for human developers.

While previous research has extensively explored LLMs, this paper delves deeper into the specifics of ChatGPT, examining its strengths and weaknesses. Our objective is to pave the way for a future where LLMs automatically fix any bugs in the code without any human assistance. This necessitates a comprehensive understanding of ChatGPT's capabilities and the potential improvements needed to achieve this ambitious goal.

3 Background literature

In this section we will delve deeper in the literature concerning automated program repair, language models, and OpenAI's code completion model.

3.1 Automated Program Repair (APR)

Automatic program repair (APR) is the use of algorithms and techniques to automatically generate patches for bugs or defects in software programs. The goal of APR is to reduce the time and effort required to find and fix defects in software, which can save development teams time and resources, and improve the reliability and stability of the software. In addition, the patch should be comprehensible for the developers since it is vital for the maintainability and readability of the code.

Fundamentally, APR can be considered a search problem. In the process of generating a repair patch, the APR system must search through a space of potential patches to find one that fixes the defect without introducing new problems. Every candidate patch is combination of possible modifications on the source code. This search can be performed using a variety of algorithms and techniques, such as genetic algorithms, evolutionary algorithms, or constraint-based search. The specific approach used will depend on the characteristics of the defect and the program being repaired. However, there are some problems with this

approach.

First, *search space explosion* is a phenomenon that occurs when the space of possible solutions to a problem becomes too large to search effectively. It is particularly common in APR, where the search space includes all possible patches that could be used to fix a defect [15]. As the size of the program and the complexity of the defect increase, the number of possible patches also increases, making it difficult for the APR system to effectively search for a good solution. Further, it is possible that the search space used by an APR system does not contain the correct patch. This can happen because the APR system is not able to effectively search the entire space of possible patches, or if the space of possible patches is too large to search completely. In these cases, the APR system may fail to find the correct patch and instead return a patch that is only a partial or approximate solution to the defect. This can reduce the effectiveness of the APR system and lead to incomplete or incorrect repairs. To address search space explosion, APR systems may use a variety of techniques, such as restricting the search space, applying heuristics to guide the search, or using more efficient search algorithms.

Second, it is challenging to generate patches that do not introduce new defects, which was mentioned earlier as *software regression*. To avoid software regression, it is important to test the repair patch thoroughly to ensure that it fixes the defect without introducing new problems or changing the behavior of the program in unintended ways. This can be done using a variety of techniques, such as unit testing, integration testing, or regression testing. Unit testing involves testing individual components or units of the program to ensure that they are functioning correctly. Integration testing involves testing the program as a whole to ensure that all of its components are working together properly. Regression testing involves running a suite of tests on the program to ensure that the repair patch has not caused any previously fixed defects to re-emerge. By using a combination of these techniques, it is

possible to thoroughly test the repair patch and avoid software regression.

A family of techniques widely used in APR is called test suite based repair. A test suite, consisting of input/output pairs, specifies the program's functionality. It can be used as a test oracle to drive the search for the correct patch. The oracle can be divided between the *bug oracle* that exposes the bug in the code, and the *regression oracle*, which encapsulates the required functionality of the code. The latter is an essential part of the suite because the model could delete faulty code to pass all the test cases without the regression test. Further, the inputs to test suite based repair techniques are the buggy program and a test suite, which contains some passing tests as the specification of the expected behaviour and at least one falling test as a specification of the bug to be repaired. The model attempts to generate as output a patch that passes all the test cases.

Patches that pass all tests in the test suite are called *plausible* patches. The *plausible* patches are not the final product. First, these patches will undergo a manual quality inspection by developers. The developer chooses then which patches fulfil all requirements such as readability, maintainability, and efficiency. The patches that pass the final inspection are the *correct* patches (i.e. semantically equivalent to developers' produced patches).

However, there lies a problem with the test based approach. A test suite might not encapsulate all necessary test cases for a complete program specification. The consequence is that a patch might pass all the tests, but it would not fix the bug or cause software regression. The problem patches are the *plausible*, but *incorrect* patches. These patches harm the source code since they might introduce new undetected bugs or undetected regression. Furthermore, This phenomenon is called *patch overfitting*. A reason for *patch overfitting* is that *correct* patches are sparse in the search space and vastly outnumbered by incorrect patches [15]. So, it is not an easy problem to tackle. Nevertheless, there are ways to tackle overfitting. Those solutions we will discuss in the Patch evaluation section.

A typical APR process consists of 3 steps: 1) Fault Localisation, 2) Patch Generation, and 3) Patch Evaluation. In the rest of this section, we discuss these steps in more detail. Furthermore, in figure 2, we have organised some of the essential terms into a diagram for ease of understanding.

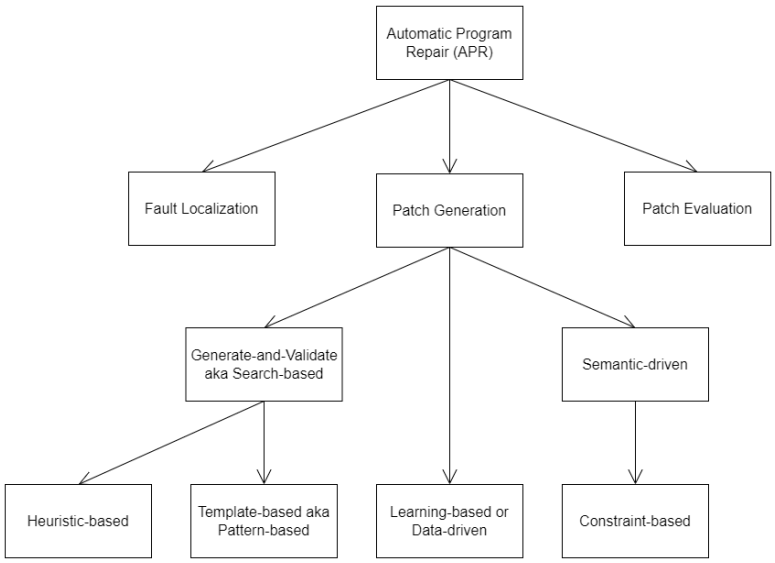


Figure 2: Terminology Tree

3.1.1 Fault Localization

To fix a bug, we first have to find its origin. Even though compilers often have an error logging system, the information in this log can point to the wrong location in the code. The reason is that it provides the point of failure, which is different from its faulty origin. For example, a variable gets assigned an incorrect value much earlier than the provided line in the error log. So, finding the bug is a big expensive part of the debugging process and largely depends on the developer's knowledge and skill set.

Fault localisation techniques aim to localise potentially faulty code elements. Fault localisation originates from different disciplines. These techniques usually analyse various

dynamic execution information to compute each program element's suspiciousness (i.e. the probability of failure). Next, it can rank the found elements based on their probability for the developers or be used for the next step, patch generation. Researchers developed several techniques to identify buggy lines, such as static analysis, code commit history, and analysing unit test case. Please refer to these surveys for more details[16, 17].

3.1.2 Patch Generation

The buggy elements are modified based on transformation rules to generate various new program variants. These program variants or patches are the candidates that will be tested to discern whether it is the correct solution. This part of the process is the most challenging because it concerns automating the human task of bug fixing. We have already mentioned it before that we cannot brute force it because of the search space explosion. There are multiple approaches for generating patches, such as using templates to transform the buggy code and generate candidate patches. We will discuss the different classes of generation techniques in next section.

3.1.3 Patch Evaluation

Patch evaluation is essential part of the process because incorrect but plausible patches harm the bug-fixing process. In other words, these incorrect plausible patches can cause cause *software regression*. For example, a possible strategy for patch generation is to delete the buggy lines of code. This strategy can pass the test suite, but it is, in the end, undesirable for the human developers. Therefore, the goal of patch evaluation is to filter out the incorrect patches so that the correct ones remain.

We already explained the patch overfitting problem previous section, but it is most relevant for patch evaluation. The key task of patch evaluation is to discern the *correct* patches

from *incorrect*. A common practice in the program is to employ manual assessment for generated patches to assess their correctness. A patch is correct if and only if: 1) it is identical to a human-written patch, or 2) the patch is semantically equivalent. Otherwise, the model shows overfitting. A key metric to determine if models is overfitting, is precision. The precision in this context is the fraction of correct patches out of all the plausible patches.

However, manual assessment has three significant problems: difficulty, bias, and scale. For the first problem, it is difficult to determine what is semantically equivalent during the repair. Therefore, a developer needs the necessary expertise considering the code base. They have to know the code requirements, whether a solution is scalable, and the coding standards for the code. Second, the person who executes the manual inspection is often responsible for developing the code. So, this practice introduces bias in the inspection process. Finally, the third problem is scale. The bigger the system becomes, the more bugs need a manual inspection, and the number of bugs can outgrow the available resources.

Patch assessment is essential because it determines the effectiveness of fault localisation and patch generation techniques. A possible solution to combat the previously described problems is automatically detecting correct patches. These are the Automated Patch Correctness Assessment (APCA) techniques. These techniques allow researchers to test at scale and reduce bias. APCA techniques can be either static or dynamic. Static techniques prioritise or filter out incorrect plausible patches by statistically analysing patches' characteristics (e.g. Anti-patterns [18]). Dynamic techniques generally leverage automated test generation tools to identify correct patches among plausible ones (e.g. DiffTGen [19]).

3.2 Patch Generation Techniques

In this section, we will discuss three different patch-generation approaches. How do they work? What advantage has an approach compared to others, and which challenges are they

currently facing?

3.2.1 Search-based

A typical patch generation technique is Generate-and-Validate (GV) (aka search-based repair), which iteratively creates candidate fixes, compiles them, and runs these candidates against the given tests. The intuition behind it is to generate possible patches and test them to see if they are plausible.

GV consists of two main techniques: heuristic-based and template-based. Heuristic-based repair technique uses a mutation operator to explore the search space, representing all possible modifications to the source code [20, 21, 22].

Heuristic repair generates patches by transforming the Abstract Syntax Tree (AST). An AST is a tree representation of the code. See Fig 3 for an example of an AST. ASTs express program structure at multiple levels of abstraction or granularity. One of the first heuristic methods is GenProg[20]. GenProg uses *genetic algorithms* to traverse the search space. The fitness of every candidate is the number of 'faults' repaired plus the number of essential functions that remain. The number of possible modifications grows exponentially, so that is where the heuristic comes in to guide the search process.

The other primary technique is template-based (aka Pattern-based). This technique generates patches by transforming a buggy program using fix patterns/transformation templates [23, 24, 25, 26, 27, 13, 28, 29]. The trade-off of templates is readability and naturalness instead of repair space size. The APR research community views template-based techniques as the current state-of-the-art method among the traditional methods.

There are multiple ways to create these patterns. The first option is to create the fix patterns manually. The advantage is that it is precise. On the other hand, this method is prohibitive and error-prone. The second option is to infer patterns by mining them from

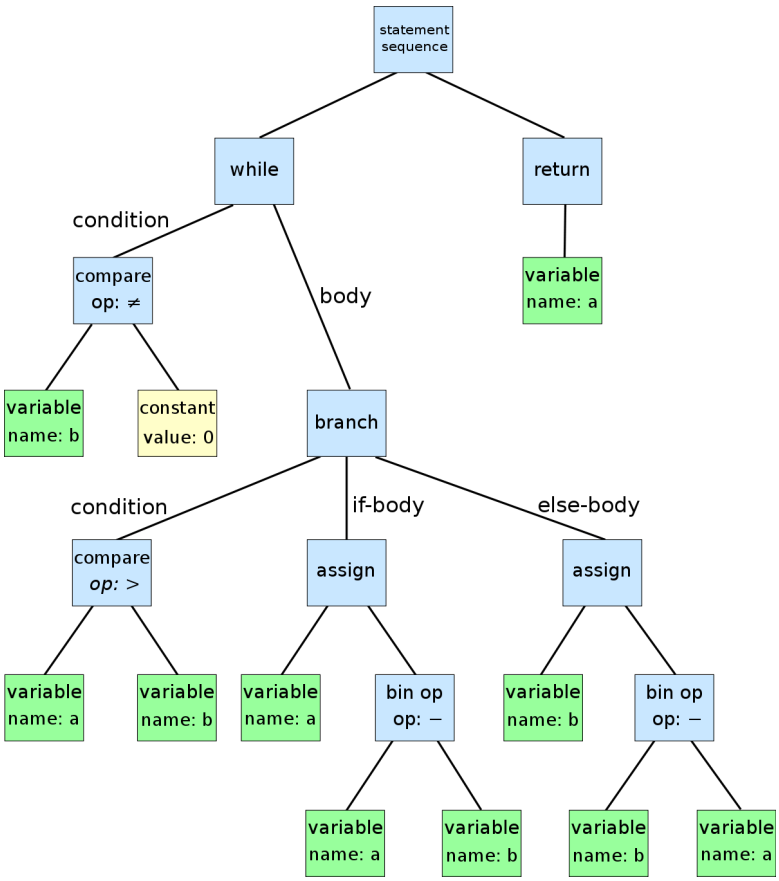


Figure 3: Example of an Abstract Syntax tree from `https://en.wikipedia.org/wiki/Abstract_syntax_tree`

existing patches[24]. There have also been other mining methods [28, 26]. Mining methods result in a large number of fix patterns, which are sometimes trivial as an operator. For example, the pattern only changes code style and its behaviour. The third option is using pre-defined patterns. For example, unify fix patterns proposed in previous studies. The fourth and last option is to infer patterns from statistics. Choose the patterns frequently used in existing patches. For example, you could take the top-n most frequent code changes as fix patterns.

However, The GV approach has two significant limitations. First, there is always the possibility that the correct patch is outside the search space. The reason for this is that

the approach has the assumption that the fix is in the code. However, finding a solution for the bug is only possible if this is the case. The solution to this could be using more modification operators. However, this gives rise to the second limitation. If there are too many transformation operators, the *search space explosion problem*[15] arises. The search space explosion problem happens when an ample search space contains many candidate patch templates, but it is infeasible to validate them all. In other words, it is computationally too costly to consider all the possible solutions.

3.2.2 Constraint-based

Constraint-based techniques have a different approach to generating patches. Instead of generate-and-validate, it first sets constraints that the patched code should satisfy and then uses them to synthesise a solution. Program synthesis is the task of constructing a program that provably satisfies a given specification. It is different from program verification since the program is a work in progress.

In detail, these techniques create a repair constraint for every test in the test suite. Next, the method synthesises a solution from the disjunction of all found conditions. The method obtains a solution to the constraints by constraint solving or other search techniques. The key to the constraint-based approach lies in the constraint formulation rather than the process of solving the constraints.

A more concrete example of the constraint-based approach is Nopol, a conditional statement repair [xuan 2018 nopol]. Conditional statements are the perfect fit for constraint-based techniques since all statements return either true or false. Additionally, if-conditions are among the most error-prone elements in programs [source Martinez 2013].

Symbolic execution analyses a program to determine what inputs cause each part of a program to execute. It uses symbolic values instead of the actual input values; thus, it

creates an expression in terms of those symbols.

However, the problem with constraint-based techniques is that it overfits to test suite. So, the methods cannot perform well on bugs not present in the test suite.

3.2.3 Learning-based

The uprise of advanced machine learning has caused waves in the scientific community, and APR is no exception. The power of deep learning combined with a large number of patches enables learning-based techniques to generate fixes for bugs [30, 31, 32, 33, 34, 35, 36, 14, 37]. An important note is that some of the previous patch generation techniques are enhanced by machine learning, but not per se learning-based because in those cases it is not the core part of the technique. Learning-based techniques typically view the problem of APR as a Neural Machine Translation (NMT) problem and use NMT techniques from the field of Natural Language Processing (NLP) to improve the performance of APR models. Consequently, NMT systems have recently achieved state-of-the-art performance on program repair tasks.

NMT is commonly used for translation between natural languages. For example, a model that translates French into Swedish. In the case of APR, it is translating buggy code into (potentially) correct code. These models typically consist of encoder-decoder architecture. Encoding means converting data into a required format, in our case, turning buggy code into an intermediate representation. To decode means to convert a coded message into intelligible language. So, the decoder turns the intermediate representation into a candidate patch.

There are currently three different approaches of learning-based techniques. The first method is to train a model by feeding it a large dataset of correct code. So, the model can provide a probability of how likely a patch is concerning the dataset. Next, the algorithm sorts the candidate patches based on realism. In other words, would a human developer

choose this patch as the correct code? An example of this is *Prophet*[33].

The second approach finds patterns for code transformation templates from successful patches in commit histories. The templates then generate the candidate patches to fix the source code. At last, the third strategy aims to create an end-to-end repair model. Thus, the model takes the buggy code as input and returns a correct bug fix. The advantage of this method is that it does not depend on a test suite. Nevertheless, we need the buggy code and its correct patch for model training.

The advantage of NMT is that it can learn complex relations between input and output sequences that are challenging to discover manually. Another advantage is that the method is not language specific. In the other APR techniques, algorithms must be redeveloped if used for another programming language.

The critical challenges of the NMT systems are:

1. **Programming Syntax**: The misuse of words can be fatal in the programming domain. Generally, with NLP tasks, a mistaken word is less costly. The reason for this is that a human can interpret from the context what the wrong word should be. A simple example is the generated sentence, ”The sky is yellow.”. A human can infer from the context that it should be blue and continues reading. However, a compiler will raise an error and stops if the source code contains the wrong syntax for that language.

2. **Context Representation**: Dependencies in a code are often not in the same line or block. In natural language, the context is generally in the same sentence or within a couple of sentences. On the contrary, a program can define a global variable at the beginning and use it at the end. Deep learning models such as recurrent neural networks (RNN) have trouble with long-term dependencies.

3. **Noisy data**: It is challenging to find commits that focus solely on a bug fix.

4. **Large vocabulary**: The vocabulary of programs is larger than a natural language corpus because developers are not limited by vocabulary to choose names for variables and functions. For example, a function that returns the lowest number from a list can have different names and spellings, such as Min(), Minimum, choose_min_from_list(), getMin(), and getLowestInt(). The large vocabulary leads to the curse of dimensionality. There is not enough data for all combinations, which causes the model to generalize poorly.

5. **Uncompilable patches**: NMT models often generate patches that do not compile. This phenomenon is a massive waste of resources.

6. **Repeated patches**: The model often optimizes to produce identical patches. This phenomenon is the consequence of the cross-entropy loss function often in deep learning models. However, the goal is to generate semantically equivalent patches.

Even though these multiple issues, there already have been solutions for parts of these challenges. For example, Chen et al.[30] have a solution for the fourth issue. They utilize a copy mechanism to deal with unknown tokens. The model copies unknown tokes from the input to the output. So, in a translation task, the model does not know ”London” and then copies the London token to the output, which is correct because it is not necessary to translate location entities. Another solution for a problem is CURE's approach[32]. The researchers use pre-trained models to learn code that can compile, and so solving problem number five.

The interest in learning-based approaches for software engineering problems keeps increasing. Thus, we will see much more progress solving the challenges in this domain in the coming years.

3.3 ChatGPT

In this paper, we will evaluate the ChatGPT model developed by OpenAI. ChatGPT was initially released on November 30, 2022, as a "research preview" and is based on the GPT-3.5 series. However, it is now possible to run it using the more advanced GPT-4 model. ChatGPT is closely related to InstructGPT, which is specifically trained to follow instructions and provide appropriate responses. In contrast, ChatGPT is designed to understand and generate human-like answers in response to various prompts.

The remarkable ability of ChatGPT to produce human-like and often accurate responses to a wide range of questions has catapulted it to fame as the fastest-growing app of all time. Within just two months, it amassed an astounding 100 million users worldwide. Notably, one key distinction between InstructGPT and ChatGPT lies in their manner of prompting. ChatGPT's conversational style allows it to challenge incorrect premises and even admit its mistakes, fostering a more interactive and dynamic user experience.

The researchers of OpenAI trained ChatGPT using Reinforcement Learning from Human Feedback (RLHF). The goal of RLHF is to make the model safer, more helpful, and aligned. RLHF uses human feedback as a reward function, and therefore get the desired behaviour. The advantage of this method is there is no need for a human researcher to develop a goal function. Designing a goal function is often an sub-optimal approximation since humans have trouble putting their intuitions of correct behaviour into formal rules. RLHF bypasses this formal definition step. The AI gradually builds a model of the goal of the task by finding the reward function that best explains the human's judgement.

The training process consisted of three steps. In the first step, a model is trained using supervised learning. The labeled data is created by retrieving a prompt from their database and consequently an AI trainers demonstrates the desired output behaviour. This data fine-tunes the GPT-3.5 model. The next step involves giving the new fine-tuned model prompts

and generate multiple outputs. The output of the model is then ranked by an AI trainer from best to worst and this data is used to train a reward model. The final step is to fine-tune the GPT-3.5 model using this reward model. The model generates a prompt and then the prompt is reward according to the reward model. The reward is used to update the model. This process is repeated for multiple iterations. According to OpenAI, these are some of the limitations of ChatGPT[38].

- ChatGPT writes sometimes plausible-sounding but incorrect answers.
- ChatGPT is sensitive the small differences between input phrasing or even attempting the same prompt multiple times.
- The model has trouble to be concise. It often overuses long sentences.
- The model guesses the intention of the user instead of asking for clarification.

4 Experiment Setup

4.1 Research Questions

Our evaluation aims to answer the following research questions:

- **RQ1: How effectively does ChatGPT fix real world bugs?** To answer this question, we evaluated our approach on the widely used APR benchmark, Defects4j v2.0.
- **RQ2: Can ChatGPT outperform other existing techniques?** To answer this question, we compare the results to other APR techniques.

4.2 Dataset

For the evaluation of ChatGPT, we have selected the Defects4j benchmark dataset as our primary resource[39]. Defects4j stands out as an ideal choice due to its wide usage and its ability to facilitate direct comparisons with other state-of-the-art APR techniques. It comprises a diverse collection of reproducible bugs written in Java, accompanied by a robust infrastructure that supports advancements in software engineering research.

Defects4j offers several features, one of which is its user-friendly interface for compiling and executing test suites. This ease of use streamlines the evaluation process and ensures a smooth workflow. Moreover, Defects4j has evolved over time, and the current version 2.0 (as shown in Table 1) boasts an impressive expansion, encompassing a total of 835 bugs. This significant increase from the original 1.2 version, which had 395 bugs. This expansion broadens the scope of our evaluation and allows for a more comprehensive analysis of ChatGPT's performance.

While the augmented size of the benchmark dataset is notable, it is important to acknowledge that not all bugs within Defects4j can be included in our experiments. Specifically, we

Project	Bug Count	In v1.2?
Chart	26	Yes
Cli	39	No
Closure	174	Yes
Codec	18	No
Compress	47	No
Csv	16	No
Gson	18	No
JacksonCore	26	No
JacksonDataBjnd	112	No
JacksonXml	6	No
Jsoup	93	No
JxPath	22	No
Lang	64	Yes
Math	106	Yes
Mockito	38	Yes
Time	26	Yes
Total	835	

Table 1: Defects4j v2.0 projects and their respective bug counts.

encounter limitations with bugs that consist of multiple hunks, which our current pipeline is unable to handle. Our pipeline cannot handle these type of bugs, because of the token limit on the API. Thus, for the purpose of this evaluation, we will focus on the evaluation of single hunk bugs. This approach ensures a more accurate assessment because its a limitation of the API and not the ChatGPT model itself.

To ensure a fair comparison between APR techniques, we will only consider the bugs that were addressed during the experiment by both ChatGPT and the alternative techniques under scrutiny. By doing so, we compare the effectiveness and performance of ChatGPT within the context of Defects4j.

Furthermore, in our original plan, we intended to incorporate additional benchmarks into our experiment. The inclusion of a wider variety of datasets offers the advantage of improving the generalization of our results by reducing the potential for test suite overfitting.

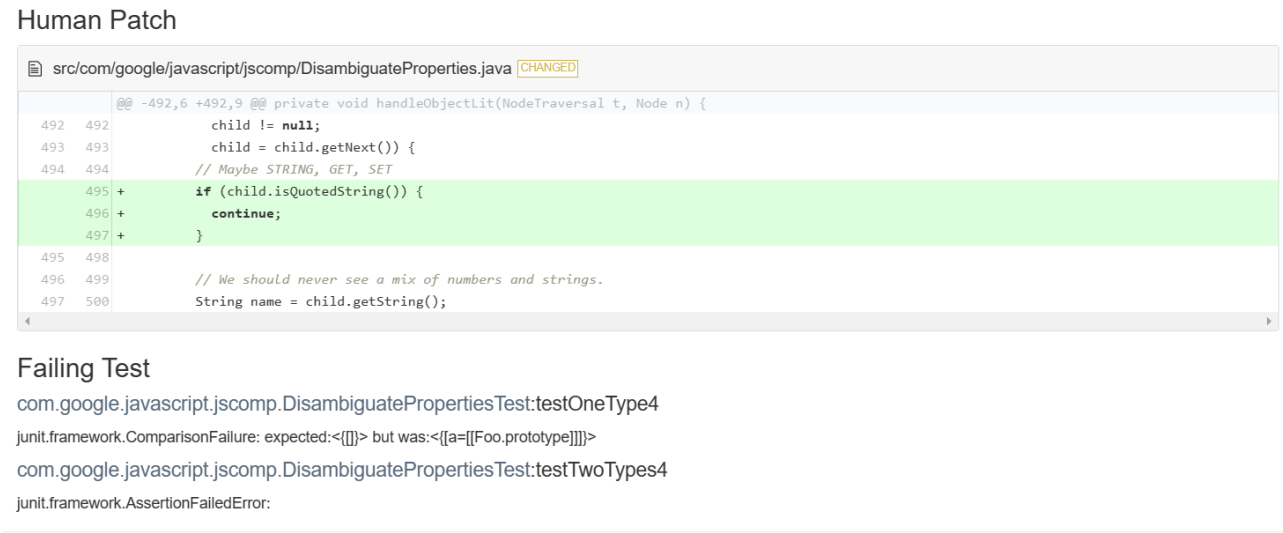


Figure 4: Example of a patch in Defects4j `http://program-repair.org/defects4j-dissection/#!/bug/Closure/118`

Two datasets that we considered for inclusion are Quixbugs[40] and bugs.jar[41].

Quixbugs presents an advantage over Defects4j as it consists of problems written in Python, a language in which ChatGPT is known to excel. The reason is that a significant part of the code on GitHub (ChatGPT's training data) is written in Python. However, one drawback is that Quixbugs is relatively less popular compared to Defects4j. Consequently, the availability of alternative techniques for comparison purposes would be limited.

On the other hand, bugs.jar shares the disadvantages of being both unpopular and written in Java. However, we view this dataset primarily as an extension of the test dataset, enabling enhanced generalization of results.

Unfortunately, we were unable to incorporate Quixbugs or bugs.jar into our evaluation, limiting our analysis to the Defects4j benchmark.

4.3 Fault Localization

In our experiment, we had to make a certain assumption to evaluate ChatGPT. Due to limitations in the size of the prompt and response, we were unable to send the entire faulty repository as a prompt. Instead, we narrowed down the scope and provided ChatGPT with only the faulty function. This approach, known as perfect localization, assumes that ChatGPT will always receive the faulty function for determining its performance in patch generation, rather than evaluating the complete bug fixing process.

4.4 Prompt Engineering & Hyperparameter tuning

Prompt engineering plays a vital role in our experiment, considering the inherent variability of natural language as input. While this variability presents both advantages and challenges, it emphasises the importance of constructing effective prompts. On one hand, the flexibility of input allows us to generate responses for a wide range of queries. However, it also introduces a degree of randomness, making it necessary to undergo a trial-and-error process to filter out poor-quality results or incorrect intention alignments. Even when narrowing down the prompt to a specific task, such as "Fix my code," the various possible phrasings can yield different outcomes.

In our approach, we consider prompt engineering as part of the hyperparameter tuning phase since the prompt serves as one of the parameters to fine-tune the model. To evaluate the performance of prompts and the conventional hyperparameters, we created a small test subset from the Defects4j dataset, comprising the first 10 bugs from projects like Chart, Closure, Compress, Csv, Gson, Lang, Math, and Time. These 80 bugs serve as a benchmark to assess the impact of prompts and hyperparameters.

Among the key hyperparameters, we focused on the *temperature*, *top p*, and *best of*. The

temperature parameter controls the level of randomness in the model's output. As the temperature approaches zero, the model becomes deterministic and repetitive, while higher temperatures lead to more exploratory behavior. The *top p* parameter also influences diversity, and OpenAI advises against using both parameters simultaneously. Consequently, in our tuning phase, we utilized either the *temperature* or the *top p* parameter. Additionally, the *best of* parameter determines the number of samples the model generates before returning the best response. For instance, a *best of 5* setting selects the best response from five generated samples. However, due to a long repair time per bug, we set this value to 1, resulting in generating the best (highest log probability) response according to the model.

We have used grid sampling to test 5 prompts and 10 model values, which results in 50 different combinations. Out of the 10 model values there were five values for the temperature and five values for the top_p., since these were discouraged to be used at the same time.

Our resulting prompt engineering approach followed a direct instruction style, utilizing imperatives to convey the task to the model effectively. Figure 5 illustrates the final prompt, with the yellow lines remaining consistent while the green lines change depending on the specific problem. Including the "Do not add anything" line was crucial to prevent the model from generating code beyond the function scope. Through experimentation, we determined that a temperature setting of 0.8 yielded the highest performance among the hyperparameters examined.

4.5 Experiment

The experiment consists of three phases: generation, validation, and evaluation. In the generation phase, we utilize the Defects4j benchmark and the ChatGPT API to generate patches for the benchmark problems. The prompt will be added to the content of the API request

```
#### Fix bugs in the below function
#### Do not add anything at the end of the code
### Buggy Java
/**
 * Returns a paint for the specified value.
 *
 * @param value  the value (must be within the range specified by the
 *         lower and upper bounds for the scale).
 *
 * @return A paint for the specified value.
 */
public Paint getPaint(double value) {
   double v = Math.max(value, this.lowerBound);
   v = Math.min(v, this.upperBound);
   int g = (int) ((v - this.lowerBound) / (this.upperBound
         - this.lowerBound) * 255.0);
   return new Color(g, g, g);
}
### Fixed Java
```

Figure 5: Example of a prompt for ChatGPT.

and the API will send a message back containing the generated response. To overcome the token limit of ChatGPT, which has a maximum of 4096 tokens, we adopt a strategy where only the function definition (comments above the function are included) is provided as the prompt. The bug location is identified using code line numbers obtained from a program developed by René Just, the founder of Defects4j. The Javalang Python library uses the bug line number to parse the specific function and its documentation from the source code. Subsequently, the prompt is constructed and sent to the ChatGPT API, and the edited function in the response replaces the original function in the faulty source file. Finally, the framework tests the plausibility of the generated code by checking if it passes all the tests in the test suite.

At the end of this phase, we obtain a subset of patches that are both compilable and plausible. Compilable patches refer to those that can be executed without any error messages appearing in the console. To validate the patches, we utilize the test suite provided by the Defects4j framework. The validation process involves three steps. Firstly, we check if the patch is compilable by running the code. Secondly, we verify if the code passes all

previously failed test cases, focusing on the essential requirement of successful patching. Finally, we ensure that the patch does not cause software regression by examining if it passes all previously passed tests. The subset of generated patches that successfully pass all three steps are classified as plausible.

In the evaluation phase, the focus shifts to assessing the correctness of the plausible patches. Due to the complexity of the task, manual inspection is employed as APCA (Automated Program Code Analysis) techniques currently lack the sophistication to solely determine correctness. However, the Defects4j framework has the patched versions as a previous commit. These were the bug fixes provided by human developers. So, we can compare the fixed version with the plausible version if it is correct. If a patch is deemed to be of sufficient quality, it is labeled as correct. This evaluation phase also provides an opportunity to derive qualitative insights, such as examining the different approaches taken by the model to fix a bug and identifying any patterns of difficulty encountered by the model across different types of bugs.

5 Evaluation

Our experiment focuses on assessing the patch generation performance of ChatGPT. To evaluate this, we generate five patches per bug, considering both time and cost limitations. If any of these five patches is correct, we consider it a success. Additionally, we document metrics such as plausibility and compilability for each patch. The experimental results of ChatGPT are summarized in Table 2.

Project	**Correct**	**Plausible**	**Compiled**	**#Attempted Bug Fixes**
Chart	4	5	16	19
Cli	0	1	16	22
Closure	1	4	61	88
Codec	1	2	6	11
Compress	1	6	28	30
Csv	0	2	9	10
Gson	0	2	7	12
JacksonCore	1	4	14	17
JacksonDatabind	1	8	40	62
Jsoup	2	4	27	47
JxPath	0	0	6	9
Lang	4	6	25	43
Math	3	8	40	71
Mockito	0	1	8	20
Time	0	1	1	15
Total	18	54	308	476

Table 2: Results of the experiment on Defects4j v2.0.

However, it is important to note that we had to exclude some bugs from the benchmark. These excluded bugs were multi-hunk, and our pipeline does not currently support handling such cases. There are two primary reasons for this exclusion: first, the API's token limit restricts us from including everything in a single prompt, and second, we believe it is more important to prioritize less complex single-hunk bugs initially. We believe in taking

incremental steps, starting with simpler tasks before tackling more complex ones.

Furthermore, certain bugs caused the pipeline to crash because the parser failed to retrieve the faulty function. This occurred when the parser encountered certain layouts of source files and had difficulty correctly processing the functions within them. Consequently, we had to exclude these problematic cases from consideration.

Out of a total of 476 bugs, ChatGPT was able to correctly fix only 18 of them, resulting in a success rate of 3.8%. The number of plausible patches is higher but not necessarily better. The average time for a single attempt is 39 seconds, which is the reason for the maximum of 5 attempts.The precision, which measures the percentage of correct patches among the plausible ones, stands at 33.3%. If we compare this with other techniques (see table 7 in Liu et al.[13]), then the performs of ChatGPT is average.

Upon analyzing ChatGPT's responses, we made several observations. Firstly, ChatGPT tends to be reluctant to add new lines of code. Most of the correct fixes involve replacing only a few lines. Particularly, when the problem requires adding a new block to catch an exception or an edge case, ChatGPT struggles. This phenomenon is an advantage for bugs, for which the solution is a single line replacement, because the limit on the variation causes a higher compatibility rate. Secondly, we noticed that there were instances where ChatGPT was close to the solution but fell short. For example, in Figure 6, you can observe a case where the fix generated by ChatGPT is very similar to the human patch, with the exception of line 858. While the model had a similar intent as the human patch, it made an incorrect function call.

Given the proximity of this example to the correct solution, we reran the specific problem with an increased number of attempts, from 5 to 100. However, even with the increased attempts, ChatGPT did not find the correct patch. It generated numerous variations of functions on line 858 but failed to produce the correct syntax for Java. While increasing the

number of attempts might eventually lead to a solution, it would significantly impact efficiency. Considering the already limited efficiency, we opted to keep the attempts at five for now. Additionally, it's important to note that the example problem mentioned earlier was relatively simple, involving only two lines of code. In cases where problems are more complex, such as involving multiple lines or locations, the number of possibilities increases exponentially. In such scenarios, increasing the number of attempts becomes even more wasteful.

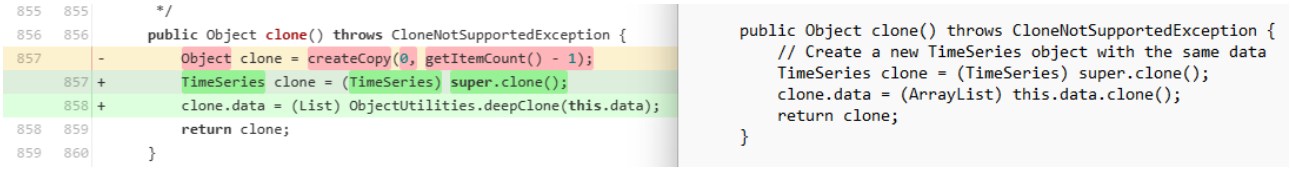


Figure 6: Example of a response which has a correct intention, but the wrong syntax.
Left: Human Patch. Right: ChatGPT response

If we look at the compile numbers in the table then we could see that 35.3% of all bugs does not have even one response that could be run. The observation that we could make from the response is that the model returns lines with impossible function calls (see previous example 6). Even though, developers with no knowledge of Java could assume it to be compilable. However, it is difficult to compare to other techniques, because almost all of them have more than 5 attempts per bug. For example, the CURE paper[32] contains a table with compile rates for different APR techniques. The problem is that the table starts with top 30 candidates, which were not able to mirror. Further,

We compare ChatGPT with six APR techniques that also performed a experiment with Perfect Fault Localization [14]. Table 3 shows the comparison results. ChatGPT performed the worst of them all. The probable cause of this result is the lack of attempts, which we discussed earlier. Another possible cause we have already mentioned before is that we were not able to test on all bugs in the respective projects. For example the Closure project has

Project	**ChatGPT**	SequenceR	Codit	DLFix	CoCoNuT	Tbar	Recorder
Chart	4	3	4	5	7	**11**	**11**
Closure	1	3	3	11	9	17	**26**
Lang	4	3	3	8	7	**13**	10
Math	3	4	6	13	16	**22**	19
Time	0	0	0	2	1	2	**3**
Mockito	0	0	0	1	4	3	2
Total	12	13	16	40	44	68	**71**

Table 3: A comparison of the number of correct patches with Perfect Fault Localization (Zhu et al, 2021)

174 bugs. ChatGPT was tested on 88 of them, so almost half compared to others. However, a consideration is that these bugs were often more complex since they were multi hunk. So, it's no guarantee that these would improve the score much, but it is still important to keep it in mind. Nevertheless, ChatGPT under performed compared to the other techniques. It lacks the sophistication in respect to knowing the syntax of the programming language. It does show promise for a general-purpose model. Especially since OpenAI has recently released the more sophisticated GPT-4 series, which could improve the results for ChatGPT by tackling the syntax problem.

6 Future Work

6.1 Limitations

The biggest limiting factor on the results is the meager number of runs per bug. Other APR techniques are able to generate ten thousands of candidates in order to find the fix compared to our five attempts. It was not possible to increase the number of attempts, because of the both time and cost. The part that is the bottleneck per run is waiting for the OpenAI API to return a response containing the generated patch. OpenAI does provide a way to get priority and speeds this up. However, the experiments will then become too expensive.

Furthermore, another limitation is the sensitivity of the model. The model does not see the code as strictly code, but as text. The importance of this distinction is that the generation is more loosely considering the constraints. In other words, the model returns often uncompilable code simply because of a syntax error. It could be for example wrong indentation, missing bracket, or function call which are not possible. This problem is a waste of resources, but there is no way to enforce these constraints from the outside. Those constraints have to be build into the model, which causes the model to lose its 'general-purpose' tag. Maybe, it needs more training/sophistication for the model to be more cost effective.

Third, there was the problem of context size. It was not possible to enter the complete file, let alone the entire repository. This means that the fault localization techniques has to do a lot of the heavy lifting. In our experiment we provided the model with the function containing the faulty lines. However, this is the ideal case and in reality it means that the performance of our model has to subtracted with the cases that our model fixes the bug, but the FL technique would not be able to find it. Another consequence of this issue is that the ChatGPT model misses essential context. For example, the faulty call on another function, ChatGPT has no possible way of knowing what this function. The only information it has is

the function name and the names of the provided arguments.

Fourth, the prompt engineering is very subjective and finicky. The way you phrase the question/demand has significant impact on the response. To illustrate, you ask the model "Fix the code below" or "Can you fix the code below". Both sentence has the same intent, but a different outcome. Furthermore, some prompts are not possible because the model returns more than just the function. It continues generating until it hits the token limit. There is a possible solution is to retrieve the function from the response, but this is another step, which introduces more fragility into the pipeline. This solution also removes one of the key advantages, which is that the model is not language specific. You have to tailor the program to the given language.

6.2 Threats to validity

Our experiment is subject to several potential threats to its validity that need to be addressed. Firstly, the popularity of the Defects4j benchmark poses both advantages and disadvantages. While it allows for comparisons with other APR techniques, it also raises concerns regarding data snooping. Given that GitHub, which was part of ChatGPT's training data, may contain numerous fixes for these benchmark problems, there is a risk that our evaluation results may not generalize well to new problems because of overfitting. Although we originally planned to test the model on other benchmarks, time constraints prevented us from doing so. Furthermore, the proprietary nature of ChatGPT's technology restricts our ability to retrain the model without the benchmark repositories.

An additional internal threat involves the assumption that the fixes provided by Defects4j framework are correct. While the patches were written by humans, it is possible that some of them may not adhere to established coding standards. However, the subjective nature of coding standards and the absence of a consensus on the best practices make it

challenging to objectively evaluate the correctness of these fixes.

Lastly, it is important to acknowledge that our experiment assumes that ChatGPT has sufficient context with only a function definition and its comments. However, this approach overlooks the dependencies and internal workings of the function. Even human developers would struggle to repair all the bugs in the benchmark based solely on a function definition. Therefore, this limitation is inherent to the approach rather than a flaw specific to the model itself.

6.3 Follow-up studies

The experiment served as an interesting starting point to assess the capabilities of ChatGPT. Moving forward, there are several avenues for future research and experimentation. One obvious area to focus on is optimizing the pipeline, which would allow for a significant increase in the number of attempts. Currently, this is the experiment's most prominent limitation. By expanding the number of attempts, a more accurate comparison between state-of-the-art APR techniques could be achieved.

Another potential approach is to explore the utilization of the *best_of* hyperparameter. This parameter enables the generation of multiple candidates and returns the one deemed "best" based on the highest log probability per token. However, at present, this option is constrained by the token limit, as generating additional candidates consumes tokens rapidly. To fully leverage the *best_of* parameter, an increase in the token limit by OpenAI would be necessary.

Additionally, experimenting with the new GPT-4 series holds promise. With its enhanced power and sophistication compared to its predecessor GPT-3.5, GPT-4 could potentially yield higher performance on the benchmarks.

Further, there is the option to experiment on different languages. A major advantage of

LLMs is their general-purpose approach. So, the model is not locked in a single domain. The possible result could be that adding more datasets could keep ChatGPT's result stable, but other APR techniques might suffer.

A further suggestion for future research lies in delving deeper into prompt engineering. By providing more detailed information without overwhelming the model, it may be possible to fine-tune its performance without overwhelming the model.

Lastly, while it may not be feasible in the immediate future, testing ChatGPT on Fault Localization (FL) would be an intriguing direction to explore, particularly if the token limit is ever increased. This would enable a comprehensive evaluation of ChatGPT as a complete bug-fixing tool.

These potential avenues for future research offer exciting possibilities to enhance ChatGPT's performance and expand its application in the field of automated program repair.

7 Conclusion

ChatGPT has garnered significant attention as its capabilities continue to be a subject of curiosity. Since its recent release, there are still numerous unexplored capabilities waiting to be discovered. One particular area that could greatly benefit from ChatGPT's potential is bug-fixing, a time-consuming process for developers. In this experiment, we assess the patch generation capabilities of ChatGPT using the Defects4j benchmark.

Assuming perfect fault localization, ChatGPT managed to fix 18/54 bugs correctly/plausibly. While these results fell short when compared to other state-of-the-art techniques, it's important to consider certain factors that may have limited the performance evaluation.

Despite these limitations, there were glimpses of potential observed during the experiment. Additionally, with the highly anticipated release of GPT-4, which promises to be even more advanced, ChatGPT could potentially emerge as a front runner in the field of automated program repair.

8 References

References and Notes

[1] Devon H O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, 2017.

[2] Michael Newman. Software errors cost us economy $59.5 billion annually. *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[3] Ahmad R Kirmani. Artificial intelligence-enabled science poetry. *ACS Energy Letters*, 8(1):574–576, 2022.

[4] Shaohua Wu, Xudong Zhao, Tong Yu, Rongguo Zhang, Chong Shen, Hongli Liu, Feng Li, Hong Zhu, Jiangang Luo, Liang Xu, et al. Yuan 1.0: Large-scale pre-trained language model in zero-shot and few-shot learning. *arXiv preprint arXiv:2110.04725*, 2021.

[5] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*, 2022.

[6] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[8] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[9] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

[10] 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*, pages 75–87, 2020.

[11] 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*, pages 19–30, 2019.

[12] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[13] 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*, pages 31–42, 2019.

[14] 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*, pages 341–353, 2021.

[15] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 702–713. IEEE, 2016.

[16] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[17] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology*, 124:106312, 2020.

[18] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738, 2016.

[19] Qi Xin and Steven P Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pages 226–236, 2017.

[20] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[21] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[22] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE, 2018.

[23] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, 2016.

[24] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.

[25] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.

[26] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE, 2017.

[27] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 888–891, 2018.

[28] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, 2020.

[29] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *International Symposium on Search Based Software Engineering*, pages 65–86. Springer, 2018.

[30] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[31] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 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*, pages 101–114, 2020.

[32] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[33] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[35] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*, 2017.

[36] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1506–1518, 2022.

[37] Jindae Kim and Sunghun Kim. Automatic patch generation with context-based change application. *Empirical Software Engineering*, 24(6):4071–4106, 2019.

[38] Konstantinos I Roumeliotis and Nikolaos D Tselikas. Chatgpt and open-ai models: A preliminary review. *Future Internet*, 15(6):192, 2023.

[39] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[40] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[41] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*, pages 10–13, 2018.