Can OpenAI's Codex Fix Bugs?

An evaluation on QuixBugs

Julian Aron Prenner
prenner@inf.unibz.it
Free University of Bozen-Bolzano
Italy

Hlib Babii
Hlib.Babii@stud-inf.unibz.it
Free University of Bozen-Bolzano
Italy

Romain Robbes
rrobbes@unibz.it
Free University of Bozen-Bolzano
Italy

ABSTRACT

OpenAI's Codex, a GPT-3 like model trained on a large code corpus, has made headlines in and outside of academia. Given a short user-provided description, it is capable of synthesizing code snippets that are syntactically and semantically valid in most cases. In this work, we want to investigate whether Codex is able to localize and fix bugs, two important tasks in automated program repair. Our initial evaluation uses the multi-language QuixBugs benchmark (40 bugs in both Python and Java). We find that, despite *not being trained for APR*, Codex is surprisingly effective, and competitive with recent state of the art techniques. Our results also show that Codex is more successful at repairing Python than Java, fixing 50% more bugs in Python.

CCS CONCEPTS

• **Software and its engineering** → *Software creation and management*.

KEYWORDS

automatic program repair, deep learning, Codex, QuixBugs

ACM Reference Format:
Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3524459.3527351

1 INTRODUCTION

Finding and fixing bugs costs billions yearly [1] and takes up a considerable proportion of developer time [15]. The field of Automatic program repair (APR) attempts to develop tools that can automatically find and fix bugs in software. Many existing APR tools follow a test-driven approach: bugs need to be exposed by a failing test case and the repaired program must pass all tests, including the previously failing ones. A variety of different APR approaches have been proposed in the recent years: i) using genetic-programming (e.g., GenProg [8] or ARJA [29]), ii) using repair patterns (such as PAR [13], ELIXIR [24] or TBar [17]), iii) code retrieval-based approaches (e.g., ssFix [26] or LSRepair [18]), iv) or using deep learning (e.g., SequenceR [4], HOPPITY [6], CoCoNuT [19] or CURE [10]).

While there has been some promising work using deep neural networks for program repair, several avenues of research are yet to explore in this area. In particular, Kaplan et al. [12] observed that Transformer-based language models are subject to several *scaling laws*, including that the performance of a language model has a power law relationship with model size, dataset size, and amount of computing power invested in training the language model, as long as none of these factors is a bottleneck. Other laws show that model performance during training is strongly correlated with out of distribution performance, and that larger models require *less* optimization steps and data points than smaller models to achieve the same performance. Taken together, these laws provide evidence for training very large language models.

One such model is GPT-3, an auto-regressive Transformer language model with 175 billion parameters, which has set a new state of the art in many human language understanding tasks [2]. Unlike previous models (such as BERT [5]) that are pre-trained on unlabeled text and fine-tuned on a target task, GPT-3's size allows it to achieve this performance *without fine-tuning on the target task*, solely through its pre-training as a language model (which consists in "guessing the next word" on a large amount of text). Adapting GPT-3 to a particular task is done in a few-shot setting by feeding a task description and a handful of examples (in most cases ranging between 3 and 10) of the task to the model at inference time, and asking the model to complete the text. In some cases, just feeding the task description without examples (zero-shot setting) shows very good performance. Thus, instead of gathering new data and fine-tuning the model on it, the user's task shifts to defining a prompt that triggers the desired behavior in the language model.

Recently, OpenAI[1] released Codex [3], a GPT-3 like model targeted towards code tasks. Codex is at the core of Copilot[2], GitHub's AI coding assistant that provides code completion in Visual Studio Code. In OpenAI demos, Codex is able to synthesize whole functions from a short description. Codex is mostly used in a zero-shot setting: the input is comprised of a short task description and a final *prompt*. Codex then generates code that "naturally" "completes" the prompt.

In this paper we investigate whether Codex can be applied to the challenging task of Automatic Program Repair. Rather than synthesizing code from natural language problem description, we ask: 1) whether Codex shows promise in repairing buggy code, a task that it was *not trained on*, and 2) which types of prompts yield the best

APR'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527351

[1]the authors are not affiliated with OpenAI
[2]https://copilot.github.com/

performance (we experiment with 5 different configurations). Since, unlike the majority of APR tools, Codex supports multiple programming languages, we evaluate performance on two programming languages. This multi-lingual requirement leads us to choose the QuixBugs [16] program repair benchmark to conduct this initial investigation in Codex's performance for APR. QuixBugs contains buggy Python and Java implementations of 40 classical computer science algorithms, such as counting bits in an integer, calculating the Levenshtein distance or finding the shortest path in a graph (see Table 2 for the full list of algorithms).

We further discuss most common model mistakes and compare repair performance between Python and Java and between Codex and previous neural repair approaches. We find that, especially considering it was not trained on the task, Codex is surprisingly competitive with three recent APR tools (CURE [10] and DeepDebug [7] released in 2021, CoCoNut [19], released in 2020), and is in the lead for Python. We have publicly released all our inputs and Codex' outputs[3].

2 BACKGROUND

2.1 Automatic Program Repair

The goal of Automatic Program Repair (APR) is to automatically fix software defects. Traditionally, program repair tools receive as input a faulty program along with a test suite of at least one fault-exposing test case. Repair is usually preceded by *fault localization*, which identifies and ranks likely buggy locations in the code. So-called *generate-and-validate* approaches employ a search-like strategy: previously localized locations are repeatedly modified until all previously failing test cases pass. In addition to that, originally passing test cases should also pass. The above-mentioned modification can be made in different ways. For instance, GenProg [8] or ARJA [29] use genetic programming while PAR [13], ELIXIR [24] or TBar [17] apply pre-defined transformation rules (repair patterns) to the code.

An alternative line of research leverages formal methods such as satisfiability modulo theories (SMT) or symbolic execution to synthesize expressions that, after being substituted into the code, make all test cases pass. This includes, for example, tools like Nopol [27], Angelix [20] or SemFix [21].

In recent years, the application of deep neural networks for APR has attracted increasing interest in the community. Different models and architectures have been explored: SequenceR [4] uses a LSTM network, HOPPITY [6] combines an LSTM with a Graph Neural Network, CoCoNuT [19] relies on a convolutional seq2seq model; finally, CURE [10] employs a GPT-based language model. Deep learning-based methods do not strictly require bug-exposing test cases for the repair process. This is a considerable advantage as recent work suggests that such test cases are rare and often nonexistent [14, 22]. Moreover, such models can be trained to jointly localize and fix bugs (e.g., HOPPITY), eliminating the need for an external fault localization system.

In this work, the model (Codex) is used in a way that does neither require running test cases during the repair process (test cases are executed for evaluation purposes only) nor requires a separate fault localization step.

[3] https://sandbox.zenodo.org/record/934361

2.2 Some Context on Codex

Codex [3] is a large deep learning-based language model developed by OpenAI. Like GPT-3, which Codex is based on, it builds on the Transformer [25], a successful neural network architecture, but, following the scaling laws, at a very large scale. Codex' size and the amount of data used to train it are unprecedented in Software Engineering: it has 12 billion parameters and was trained on 54 million GitHub repositories. Being a language model, Codex was trained to complete (partial) input in a meaningful way. Training models of the size of Codex goes far beyond the capabilities of single GPUs and can currently only be trained by large organizations or corporations. However, large language models have been shown to be able to perform a range of different tasks [23] and can, once trained, be shared and used for different purposes. Codex supports a variety of programming languages including Python, Java, JavaScript, TypeScript, and others. Compared to GPT-3, Codex is smaller, but has a larger input window (4096 vs 2048 tokens), in order to generate code from a larger context.

Similarly to GPT-3, Codex is versatile: it is capable of carrying out several different tasks, as long as the tasks can be framed as a "completion task", in a zero-shot or few-shot setting. Thus, instead of providing data and re-training the model on it, a user of Codex engages in *prompt engineering*: finding out which prompt (possibly with examples) yields the best performance for the task. It's worth noting that this shift from fine-tuning to prompt engineering is welcome, if only for the reason that fine-tuning the model is out of reach for the vast majority of people due to its sheer size. Rather than running on a user's machine, the model runs in the cloud.

Examples of tasks where Codex can be applied are: intelligent code completion, where the input is code that should be completed (a version of Codex powers GitHub Copilot, a code completion plugin for Visual Studio Code); function synthesis, where the input is a function name and a documentation comment and the output is a code snippet; code explanation, where the input is a code snippet, and the output is a comment; and programming language translation (e.g., C++ to Python), where the input is a code snippet, followed by a comment indicating the language to translate to.

2.3 The QuixBugs Dataset

In all of our experiments we rely on QuixBugs [16], a benchmark of 40 buggy algorithm implementations, including their correct versions and test cases. An important reason for choosing QuixBugs was bilingualism: all algorithms are implemented in Python and Java. This allows to compare Codex' repair capabilities between different languages. Second, the programs in QuixBugs are relatively short (9-67 lines of code [28]) and thus fit Codex' input window of 4096 tokens.

Unlike the Python versions, which contain docstrings with short descriptions of the respective algorithms, Java versions come without descriptive comments. Figure 1 shows a buggy implementation of the Euclidean algorithm, taken from the QuixBugs dataset, in both languages. As can be seen, the Python version includes a docstring describing the algorithm and specifying input and output behavior. A in-depth analysis of the QuixBugs dataset can be found in Ye et al. [28].

```
def gcd(a, b):
    if b == 0:
        return a
    else:
       return gcd(a % b, b)
"""
Input:
    a: A nonnegative int
    b: A nonnegative int

Greatest Common Divisor

Precondition:
    isinstance(a, int) and
    isinstance(b, int)
Output:
    The greatest int that
    divides evenly into a and b
Example:
    >>> gcd(35, 21)
    7
"""
```

```
package java_programs;
import java.util.*;

public class GCD {
    public static int gcd(int a, int b) {
        if (b == 0) {
            return a;
        } else {
            return gcd(a % b, b);
        }
    }
}
```

Figure 1: Buggy versions of the algorithm calculating the greatest common divisor between two integers in Python and Java. The recursive call should read `gcd(b, a % b)`.

3 METHODOLOGY

3.1 Prompt Engineering

Since the only way to interact with Codex is via the prompt that it is given, we experiment with several different input configurations. All configurations use a variation of the following template (Python version):

```
### fix the bug in the following function
<buggy function and/or docstring here>

### fixed function
```

Code only. We simply input the buggy function (without the docstring in Python) in the template.

Code with hint. To simulate a more precise bug localization, in this configuration we add hint comments. Specifically, we put the comment `### bug is here` (or `// bug is here` for Java) before any buggy code line.

Code with docstring (Python). We input the buggy code along with the original docstring, which describes the correct behavior of the function. Note that some docstrings contain examples. For Java, docstrings are not available.

Docstring only (Python). This corresponds to the default usage of Codex, in which it synthesizes a full algorithm implementation, and thus acts as a baseline.

Correct code (Python). To see if Codex would break already correct code, in this configuration we input the bug-free ground-truth program, instead of the buggy one; ideally, Codex would repeat the code unchanged. We slightly alter the input format changing the first line to `### fix` **`a possible bug`** `in the following function`, indicating that the input *might* be bug-free.

Input-output examples. For seven Python bugs that no configuration involving code (i.e., excluding the docstring-only configuration) could correctly fix, we additionally tried including input-output examples derived from the corresponding test cases as an additional specification. The input-output examples are given in a docstring-like comment and follow the general format. We include all QuixBugs test cases associated to a program, except those exceeding a certain size (120 characters), so as not to produce an exceedingly long docstring. Figure 2 shows the full prompt with such examples.

3.2 Codex Parameters

Table 1: Used Codex parameters.

Parameter	Value
Engine	davinci-codex
Temperature	0
Max Tokens	1024
Top-p	1.0
Frequency Penalty	0.0
Presence Penalty	0.0
Stop	'###', '///'

We set Codex' parameters as shown in Table 1. We did not yet systematically investigate the effect of these parameters on repair success, as our evaluation involves a significant manual step. The temperature and top-p parameters control the randomness of the model: a higher temperature or a lower top-p yield more diverse output. The Codex documentation recommends to set temperature to zero or a low value and not to vary both, temperature and top-p. We set the temperature to zero and top-p to one, which lowers diversity but ought to give high robustness. The frequency and presence penalties prevent the model from outputing the same

```
### fix the a bug in the following function
"""
Examples:
    >>> pascal(1)
    [[1]]
    >>> pascal(2)
    [[1], [1, 1]]
    >>> pascal(3)
    [[1], [1, 1], [1, 2, 1]]
    >>> pascal(4)
    [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
    >>> pascal(5)
    [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1],
    [1, 4, 6, 4, 1]]
"""
def pascal(n):
   ...

### fixed function
```

Figure 2: Prompt with input-output examples in the docstring. Function body omitted for brevity.

tokens repeatedly. Since large portions of the input (everything except the buggy line) should in fact be repeated, we do not use such a penalty.

3.3 Evaluation

For each configuration and language we manually evaluated the output of Codex, using the following procedure:

- When Codex output multiple functions we only considered the first and discarded the remaining output.
- If the output exactly matched the correct ground-truth patch, we considered it correct.
- If the output exactly matched the input (i.e., no changes have been made by the model), we considered it incorrect.
- When output was neither identical to the ground truth nor the original buggy version, we ran the associated test cases.
- If any test failed, the output was considered incorrect.
- If all tests passed, we decided whether the fix was semantically equivalent to the ground truth.

We observed outputs that passed all test cases but were semantically incorrect only for the `kheapsort` bug, where QuixBugs' test cases do not check an edge case: the tests simply check for sortedness of the program output; however, it should only be sorted up to the k^{th} largest element.

Acceptable variations. Since Codex is generating code as a language model, and is not explicitly trained for program repair, we were more lenient in a few cases. In particular, it is natural for a source code language model to try to avoid defining multiple functions or methods with the same name. Thus, if the output of Codex was a function or method with a slightly different name, but otherwise correct, this was not considered an error; we assume that post-processing could ensure such a rename is done automatically. On the other hand, in several cases, Codex simply repeated the input program (bug included); this was, of course, deemed incorrect output.

When providing only the docstring, for four graph related problems (e.g., `breath-first-search` or `detect-cycle`), Codex assumed that the attributes pointing to the next node or child nodes should be named `next` and `children`, respectively, while tests assumed it to be named `successor` and `successors`. This means tests failed despite "reasonable" output, as there is no way for Codex to know the attribute name expected by the tests. In Table 2 we marked such cases with ✗ and provided a separate total count in parentheses. We considered this as semantically correct, as this is a reasonable assumption and the proper name was not specified in the input.

4 RESULTS

4.1 Overall Performance

Table 3 compares Codex's performance with recent previous work. We report results from the literature from three recent neural APR approaches: CoCoNut [19] uses the Neural Machine Translation (NMT) paradigm of program repair, with an ensemble of CNNs, and supports multiple languages. DeepDebug [7] is a large pre-trained Transformer that also uses the NMT paradigm (Python only): the model is fine-tuned on artificially generated bug-introducing commits, and also stack traces and program context. CURE builds on CoCoNut, adding a pre-trained language model (on Java Code) and filters suggestions based on additional context from static analysis [10]. Note that the assessment for correctness was done manually and evaluation criteria might slightly differ between these works.

Codex correctly fixed considerably more Python than Java bugs (up to 23 Python bugs, only 14 Java bugs), indicating that it can handle Python much better than Java; OpenAI does state that Codex is more capable in Python than other languages. Moreover, it is intriguing to see that Codex, *without explicit training on the task*, outperforms CoCoNuT and DeepDebug on Python, and outperforms CoCoNut in Java. While CURE does outperform Codex in Java, Codex outperforms the only other multi-lingual APR tool (CoCoNut).

> Codex is surprisingly competitive with recent work, and its performance is considerably better for Python than Java.

4.2 Performance of different prompts

Table 2 provides detailed results for each bug and configuration. For many bugs, the choice of prompt matters significantly. In fact, only 6 bugs are fixed in all scenarios and all languages. On the other hand, the prompts do complement each other: only 8 bugs are not fixed by any of the prompts.

Hints are not effective. Providing a hint comment for precise fault localization was overall not effective in our experiments. For both Python and Java, some bugs were fixed only with hints, but for others adding the hint was harmful. Overall, for Java the total number of fixed bugs was the same, while it decreased from 21 to

Table 2: List of bugs that Codex could fix successfully (✓).

	Python					Java	
	code + docstr.	code only	code + hint	docstr. only	*correct* code	code	code + hint
bitcount	✓	✓	✓	✓	✓	✓	✓
breadth-first-search	✗	✗	✗	✓✗	✓	✗	✗
bucketsort	✓	✓	✓	✓	✓	✓	✓
depth-first-search	✓	✓	✓	✓✗	✓	✗	✓
detect-cycle	✗	✗	✗	✓✗	✓	✗	✗
find-first-in-sorted	✓	✓	✗	✗	✓	✓	✗
find-in-sorted	✗	✗	✗	✗	✓	✗	✗
flatten	✓	✓	✓	✓	✓	✗	✗
gcd	✓	✓	✓	✗	✓	✓	✓
get-factors	✓	✓	✓	✓	✓	✗	✗
hanoi	✓	✓	✗	✗	✓	✗	✗
is-valid-parenthesization	✓	✓	✓	✓	✓	✓	✓
kheapsort	✗	✓	✓	✗	✓	✗	✗
knapsack	✗	✓	✗	✗	✗	✓	✓
kth	✓	✗	✗	✗	✓	✓	✓
lcs-length	✓	✗	✗	✗	✗	✗	✗
levenshtein	✓	✗	✗	✓	✓	✓	✓
lis	✗	✗	✗	✗	✓	✗	✗
longest-common-subsequence	✓	✓	✓	✓	✓	✓	✓
max-sublist-sum	✓	✓	✓	✓	✓	✓	✓
mergesort	✗	✗	✗	✗	✓	✓	✓
minimum-spanning-tree	✗	✗	✓	✗	✓	✗	✗
next-palindrome	✗	✗	✗	✗	✓	✗	✗
next-permutation	✓	✗	✗	✓	✓	✗	✗
pascal	✗	✗	✗	✓	✓	✗	✗
possible-change	✗	✓	✗	✗	✓	✗	✗
powerset	✓	✓	✓	✓	✓	✗	✗
quicksort	✓	✓	✓	✓	✓	✗	✗
reverse-linked-list	✓	✓	✓	✓✗	✓	✓	✓
rpn-eval	✗	✗	✗	✗	✓	✗	✗
shortest-path-length	✗	✗	✗	✗	✓	✗	✗
shortest-path-lengths	✓	✗	✗	✗	✗	✗	✗
shortest-paths	✗	✗	✗	✗	✓	✗	✗
shunting-yard	✗	✗	✓	✗	✗	✗	✗
sieve	✓	✓	✓	✗	✓	✗	✓
sqrt	✓	✓	✓	✓	✓	✓	✗
subsequences	✗	✗	✗	✓	✓	✗	✗
to-base	✓	✓	✓	✗	✓	✗	✗
topological-ordering	✗	✗	✗	✗	✗	✗	✗
wrap	✓	✓	✓	✗	✗	✓	✓
Total	23	21	19	14 (18)	34	14	14

19 in Python. However, hints led Codex to correctly repair two bugs that could not be successfully repaired otherwise (`shunting-yard` and `minimum-spanning-tree`).

Synthesis from docstrings. Table 2 shows that only providing the Python docstrings, Codex is able to synthesize a correct solution for 45% of the problems in QuixBugs. Providing buggy code as a starting point and asking to model to fix it led to five more (+28%) correct program implementations.

Additional input-output examples. For the seven bugs that no other configuration could solve, a single one (`subsequences`) was successfully repaired when adding input-output examples from test cases.

Correct code. When requested to fix *a possible bug* in correct code, Codex broke six of the total 40 programs. In two cases, Codex slightly altered the input program, preserving correctness, however.

Table 3: Comparison with previous work. Number of correctly fixed bugs for the Java and Python version of QuixBugs (out of 40).

	Java	Python
DeepDebug [7]	–	21
CoCoNuT [19]	13	19
CURE [10]	26	–
Codex	14	23/21[1]

[1] with and without docstring, respectively

Prompts have a major effect on Codex' bug fixing ability

5 LIMITATIONS AND FUTURE WORK

Codex is a very large language model, that has shown impressive ability in completing source code. In this work, we have evaluated—and found surprisingly competitive—the performance of Codex as an APR tool, *with no further training on the task*. While this initial evaluation of Codex as an APR tool is promising, it has various limitations.

More annotators. The correctness of the code output was assessed by a single annotator. Not only is manual evaluation subjective, it is also prone to mistakes. We hope that we will be able to provide a more reliable evaluation in the future, involving at least two annotators. In many cases, however, Codex' output was either unchanged input code or matched the ground-truth, slightly limiting potential annotator bias.

Additional benchmarks and languages. Currently our evaluation is limited to a single benchmark and two programming languages. Extending this study to benchmarks that involve more complex codebases (e.g., Defects4J [11] or additional programming languages (e.g., BugsJS [9], a JavaScript benchmark) would provide additional interesting insights into Codex' repair capabilities.

Data leakage. Codex was trained on very large amounts of code; only OpenAI staff can know with certainty which repositories were included. We cannot rule out that the correct ground-truth programs were in Codex' training set. This issue is very difficult to address. There are however mitigating factors: First, if present, these versions would constitute a tiny portion of the training data (54 million repositories). Second, if the correct program versions were in the training set, so would, very likely, also be the incorrect versions, without labels of which version is correct or incorrect. Moreover, a preliminary study of Codex for GitHub Copilot, found that while the model can indeed repeat data from the training set, this was rare (less than 0.1% of the cases), concerned code that was cloned many times, and happened mostly when the context was nearly empty [30]. Finally, Codex was never specifically trained for the task of repairing or localizing bugs.

More Automation. For this study, we had to perform several manual steps to validate the correctness of the proposals. This includes removing extraneous output from Codex, and making sure the function/method name was the one expected by the tests. Automating these tasks would make the process significantly smoother.

Testing multiple outputs. Given the lack of automation, we tried a single completion from Codex for each problem and prompt. With more automation, we would be able to try multiple outputs from Codex (by increasing the temperature). Evaluating more than one output significantly increased the performance of Codex for program synthesis (from 29 up to 70% [3]), so this could help APR as well.

Fine-Tuning. While the most common use case for Codex is to use it directly after pre-training, a fine-tuning API is available for GPT-3. If such an API is made available for Codex, this would be worthwhile exploring to improve performance.

ACKNOWLEDGMENTS

We would like to thank OpenAI for providing access to their Codex model.

REFERENCES

[1] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. *Reversible Debugging Software "Quantify the Time and Cost Saved Using Reversible Debuggers"*.
[2] 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. URL http://arxiv.org/abs/2005.14165.
[3] 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, . URL http://arxiv.org/abs/2107.03374.
[4] 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, . URL http://arxiv.org/abs/1901.01808.
[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. URL http://arxiv.org/abs/1810.04805.
[6] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS. URL https://iclr.cc/virtual_2020/poster_SJeqs6EFvB.html.
[7] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. URL http://arxiv.org/abs/2105.09352.
[8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. 38(1):54–72. ISSN 1939-3520. doi:10.1109/TSE.2011.104.
[9] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszedes, Rudolf Ferenc, and Ali Mesbah. BugsJS: A Benchmark of JavaScript Bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. doi:10.1109/ICST.2019.00019.
[10] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. URL http://arxiv.org/abs/2103.00073.
[11] 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*, ISSTA 2014, pages 437–440. Association for Computing Machinery. ISBN 978-1-4503-2645-2. doi:10.1145/2610384.2628055. URL http://doi.org/10.1145/2610384.2628055.

[12] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811. IEEE Press. ISBN 978-1-4673-3076-3.

[14] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. iFixR: Bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 314–325. Association for Computing Machinery. ISBN 978-1-4503-5572-8. doi:10.1145/3338906.3338935. URL http://doi.org/10.1145/3338906.3338935.

[15] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceeding of the 28th International Conference on Software Engineering - ICSE '06*, page 492. ACM Press. ISBN 978-1-59593-375-1. doi:10.1145/1134285.1134355. URL http://portal.acm.org/citation.cfm?doid=1134285.1134355.

[16] 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*, SPLASH Companion 2017, pages 55–56. Association for Computing Machinery. ISBN 978-1-4503-5514-8. doi:10.1145/3135932.3135941. URL http://doi.org/10.1145/3135932.3135941.

[17] 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, pages 31–42. Association for Computing Machinery, . ISBN 978-1-4503-6224-5. doi:10.1145/3293882.3330577. URL http://doi.org/10.1145/3293882.3330577.

[18] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662, . doi:10.1109/APSEC.2018.00085.

[19] 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*, ISSTA 2020, pages 101–114. Association for Computing Machinery. ISBN 978-1-4503-8008-9. doi:10.1145/3395363.3397369. URL http://doi.org/10.1145/3395363.3397369.

[20] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701. doi:10.1145/2884781.2884807.

[21] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781. IEEE Press. ISBN 978-1-4673-3076-3.

[22] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi. Experience Report: How Effective is Automated Program Repair for Industrial Software? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–616. doi:10.1109/SANER48275.2020.9054829.

[23] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[24] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. doi:10.1109/ASE.2017.8115675.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. URL http://arxiv.org/abs/1706.03762.

[26] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 660–670. IEEE Press. ISBN 978-1-5386-2684-9.

[27] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. 43(1):34–55. doi:10.1109/TSE.2016.2560811. URL https://hal.archives-ouvertes.fr/hal-01285008.

[28] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, 171:110825, Jan 2021. ISSN 0164-1212. doi:10.1016/j.jss.2020.110825. URL http://dx.doi.org/10.1016/j.jss.2020.110825.

[29] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. 46(10):1040–1067. ISSN 1939-3520. doi:10.1109/TSE.2018.2874648.

[30] Albert Ziegler. Research recitation: A first look at rote learning in github copilot suggestions, 2021. URL https://docs.github.com/en/github/copilot/research-recitation.