Can ChatGPT Fix My Code?

Viktor Csuvik[1] [a], Tibor Gyimóthy[1] [b] and László Vidács[1,2] [c]
[1]University of Szeged, Department of Software Engineering, Hungary
[2]University of Szeged, MTA-SZTE Research Group on Artificial Intelligence, Hungary

Keywords: Automated Program Repair, Transformers, ChatGPT, JavaScript, Java.

Abstract: ChatGPT, a large language model (LLM) developed by OpenAI, fine-tuned on a massive dataset of text and source code, has recently gained significant attention on the internet. The model, built using the Transformer architecture, is capable of generating human-like text in a variety of tasks. In this paper, we explore the use of ChatGPT for Automated Program Repair (APR); that is, we ask the model to generate repair suggestions for instances of buggy code. We evaluate the effectiveness of our approach by comparing the repair suggestions to those made by human developers. Our results show that ChatGPT is able to generate fixes that are on par with those made by humans. Choosing the right prompt is a key aspect: on average, it was able to propose corrections in 19% of cases, but choosing the wrong input format can drop the performance to as low as 6%. By sampling real-world bugs from seminal APR datasets, generating 1000 input examples for the model, and evaluating the output manually, our study demonstrates the potential of language models for Automated Program Repair and highlights the need for further research in this area.

1 INTRODUCTION

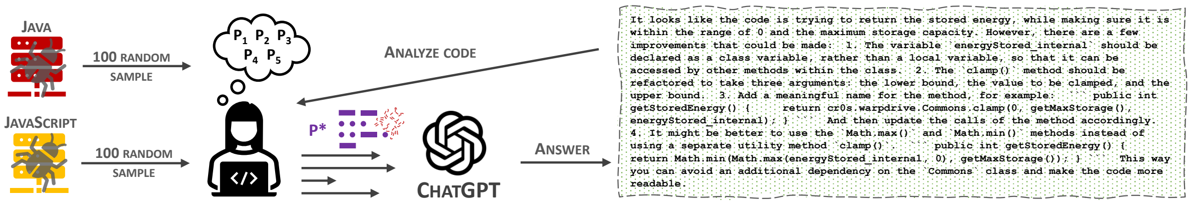
The ability of ChatGPT (OpenAI ChatGPT, 2023a) to generate human-like text and code has led to its use in news articles and personal websites, raising concerns about the potential misuse of such technology. Despite these concerns, the advancements in language models like ChatGPT have opened up new opportunities for research and have the potential to revolutionize the way we interact with computers. The urge for new and improved software encourages developers to develop rapidly, often without even testing the created source code, thus bugs are inevitable (Monperrus, 2020). The field of Automated Program Repair (APR) has emerged and gained attention in recent years. It has the potential to significantly improve software reliability and reduce the cost and time associated with manual debugging and repair (Weimer et al., 2009).

The conventional APR approach is to generate a *patch* (e.g., using genetic algorithm) and then validate it against an *oracle* (i.e., test suite). Although these approaches have been criticized several times, they still define the research direction of APR (Kechagia et al., 2022). Their standalone and easy-to-use nature makes them competitive against learning-based approaches (Liu et al., 2020). On the other hand, data-driven APR approaches utilize machine learning techniques to learn from a dataset of programs and their corresponding repair patches. These approaches often require a huge train-test-validate dataset to adapt to different repair strategies and programming languages (Yi et al., 2020; Lutellier et al., 2020). The training of such methods is resource-intensive, and the approaches are often not usable due to availability issues (e.g., confidentiality), executability concerns (e.g., specific execution environment), or configurability limitations (Kechagia et al., 2022).

As the field of Natural Language Processing (NLP) continues to advance, it is likely that we will see more applications of ChatGPT and its variants, further expanding their impact on the internet and society. In this paper, we propose an approach that fixes buggy programs automatically using ChatGPT, thus bypassing the cumbersome process of designing, training, and evaluating a new model, but simply relying on this Large Language Model (LLM). We not only explore the capabilities of ChatGPT in the domain of Automated Program Repair but also want to discuss its performance on different programming languages, and we experiment with prompts to

[a] https://orcid.org/0000-0002-8642-3017
[b] https://orcid.org/0000-0002-2123-7387
[c] https://orcid.org/0000-0002-0319-3915

Csuvik, V., GyimÃ³thy, T. and VidÃ¡cs, L.
Can ChatGPT Fix My Code?.
DOI: 10.5220/0012120800003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 478-485
ISBN: 978-989-758-665-1; ISSN: 2184-2833



```
It looks like the code is trying to return the stored energy, while making sure it is
within the range of 0 and the maximum storage capacity. However, there are a few
improvements that could be made:  1. The variable `energyStored_internal` should be
declared as a class variable, rather than a local variable, so that it can be
accessed by other methods within the class.  2. The `clamp()` method should be
refactored to take three arguments: the lower bound, the value to be clamped, and the
upper bound.  3. Add a meaningful name for the method, for example:  ``` public int
getStoredEnergy() {     return cr0s.warpdrive.Commons.clamp(0, getMaxStorage(),
energyStored_internal); } ```  And then update the calls of the method accordingly.
4. It might be better to use the `Math.max()` and `Math.min()` methods instead of
using a separate utility method `clamp()`.  ``` public int getStoredEnergy() {
return Math.min(Math.max(energyStored_internal, 0), getMaxStorage()); } ```  This way
you can avoid an additional dependency on the `Commons` class and make the code more
readable.
```

Figure 1: High-level approach of this paper.

inspect which type yields the best performance.

From previous works (Lajkó et al., 2022; Lajkó et al., 2022), we know that GPT-2 (Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, 2020) is able to generate fixes, and when it is fine-tuned, its performance is even better. It is also clear that GPT-3.5 (Brown et al., 2020) and Codex (which is essentially the same as GPT-3.5 but trained on a lot of source code) are superior compared to GPT-2 and are surprisingly effective at generating repair candidates, despite not being trained to do so (Prenner et al., 2022). To investigate the capabilities of ChatGPT, we organize our experiment along the following research questions:

RQ1: *How do various prompts perform compared to each other?*

RQ2: *What is the impact of the programming language on the repair performance?*

RQ3: *Are some bugs easier for ChatGPT to fix or not?*

We find that prompt engineering has a major role when working with ChatGPT, and different prompts tend to trigger different repair mechanisms of the model. Based on our evaluation, it seems that the proposed approach generated higher quality patches for Java and lower quality patches for JavaScript. Although the same patterns are apparent in the results, the overlap of the fixed bugs is insignificant, and we did not find easy-to-fix bug types in either of the datasets. The generated answers for each prompt, patch classifications, and examples can be found in the online repository of this paper [1].

2 METHODOLOGY

Our proposed method is quite straightforward, which we illustrated in Figure 1: ChatGPT is used with various input configurations to generate a repair patch for a buggy program. This process is illustrated in Figure 1. First, 100 random buggy programs are selected from both a Java and JavaScript database. Then, an input is formulated from the combination of this code and the prompts proposed in this work (a total of 200 distinct samples, forming 1000 inputs). Finally, the generated answer is evaluated and classified by experienced software engineers.

2.1 ChatGPT

The original Generative Pre-trained Transformer, or GPT for short, was published in 2018. ChatGPT is a descendant of this architecture. Its base is a Transformer, which is an attention model that learns to focus attention on the previous words that are most relevant to the task at hand: predicting the next word in the sentence. ChatGPT is fine-tuned from a model in the GPT-3.5 series, which finished training on a blend of text and code in early 2022. At the time of writing this paper, some details of the underlying architecture of ChatGPT are unknown, but the research community knows that ChatGPT was fine-tuned using supervised learning as well as reinforcement learning (ChatGPT: Understanding the ChatGPT AI Chatbot, 2023). In both cases, humans were involved to improve the model's performance by ranking answers from previous conversations and imitating conversations (OpenAI ChatGPT, 2023a). Although GPT-4 became available while writing this paper, in our experiments, we used the GPT-3.5 version of ChatGPT.

2.2 Dataset

APR and program generation, in general, have recently been criticized (Liu et al., 2020) because the used samples are oversimplified, and datasets are usually artificially created. Hand-crafted training data does not reflect real software defects. Thus, inputs for ChatGPT were sampled from seminal APR datasets consisting of real-world bugs. To work with multiple programming languages, we included the dataset by Tufano et al.(Tufano et al., 2019) (also included in CodeXGLUE (Lu et al., 2021)), which was created for Java program repair, and FixJS (Csuvik and Vidács, 2022), which contains JavaScript bug-fixing information from commits. From both datasets, we randomly sampled 100 instances each and formed the input using the prompts from Section 2.3.

[1]https://github.com/AAI-USZ/ICSOFT-ChatGPT

Table 1: The used prompts with example code snippets to generate a fix for a specific software bug.

`Fix or improve the following code:`	`public int nextId() { return (com.example.pustikom....studentList.size()) + 1; }`
`Modify the following Java code:`	`public int nextId() { return (com.example.pustikom.....studentList.size()) + 1; }`
`Fill in the missing part in the following code applying 78 tokens:`	`public int nextId() { ____________ + 1; } }`
`Continue the implementation of the following function using 78 tokens:`	`public int nextId() { _`
`Fix or improve the following code (with bug location hint):`	`public int nextId() { * Refinement starts here * return (com.example.pustikom.adapterplay.user.StudentList.studentList.size()) * Refinement ends here * + 1; }`

2.3 Prompts to Generate Patches

At the time of writing this paper (Q2 2023), ChatGPT is available via API and also via a graphical interface, where users can communicate with the model by inputting a prompt (OpenAI ChatGPT, 2023b). To fix a candidate buggy function, we experimented with different configurations: the input of the model consists of the sample code snippet from the observed datasets + one of the below-listed prompts. These prompts are proposed by us, but note that the choice of these is arbitrary, and we included the below ones in the paper because during our experiments, we found them interesting. We illustrated example usages of these prompts with a code snippet on Table 1. We propose the following prompts:

P_1: `Fix or improve the following code:` *`[code]`* The most natural way to prompt the model is to just input the buggy function with the instruction to fix it.

P_2: `Modify the following Java/JavaScript code:` *`[code]`* Based on our observations, the keyword *fix* or *repair* can confuse the model: it looks for a syntactical error, but in most cases, the bug is semantic - thus only refinement/modification is needed.

P_3: `Fill in the missing part in the following:` *`[code]`* `___` *`[code]`* By deleting the original buggy part of the code, we force the model to generate something in its place.

P_4: `Continue the implementation of the following function using X tokens:` *`[code]`* Since the underlying model (GPT-3) was trained to estimate the next word in a sequence, it makes sense to use the first few statements in the code and ask ChatGPT to generate the rest.

P_5: `Fix or improve the following code (with bug location hint):` *`[code]`* `* Refinement starts here *` *`[code]`* `* Refinement ends here *` *`[code]`* Essentially the same as the first prompt listed here, but here the bug location is marked with comment blocks. The specification of the exact location of a bug might not be too realistic since real-life bug localization is usually less precise, but for the sake of experimentation, it might provide interesting insights.

2.4 Evaluation

Since the goal of ChatGPT is to mimic a human conversationalist, it is in its nature that answers are long and explanatory, with a lot of natural language text. Thus, the use of standard evaluation metrics (e.g., precision, recall) is not possible. Therefore, we manually analyzed the answers and classified them into one of the following categories:

1. **Undecided:** when we were uncertain about the correctness of the response or ChatGPT was unable to generate a fix
2. **Incorrect Patch:** the output code is different from the developer patch
3. **Fix in Answer:** the generated answer contains the correct fix for the given bug
4. **Semantical Match:** the proposed fix semantically matches the one generated by a human engineer
5. **Syntactical Match:** the returned fix is the same as the developer patch, except for whitespaces

Due to space limitations, we do not include the generated answers here, but the interested reader can find them in the online repository of this paper. Note that the category *Fix in answer* does not imply semantic correctness, since ChatGPT often only highlights code snippets and does not regenerate the whole input code. It also provides technical suggestions in natural language that are meaningful for developers. For scientific correctness, we cannot state in these cases that the model generated a patch that is semantically identical to the developer fix, but rather an answer that contained the correct fix.

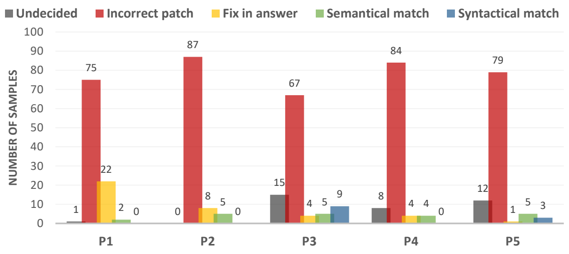


Figure 2: Manually evaluated results of ChatGPT on the Java dataset.

3 RESULTS

In this section, we present the results obtained by feeding 100 Java and 100 JavaScript samples to ChatGPT using 5 different prompts (forming a total of 1000 trials/input-output pairs). Figure 2 and Figure 3 show the manually evaluated results for Java and JavaScript, respectively. Upon examining the figures, it is evident that ChatGPT is not proficient in generating patches that semantically or syntactically match the developer fix. Instead, it provides suggestions on how to repair the code or generates a fix that contains the correct solution. The manual evaluation also revealed that different prompts trigger different response mechanisms from the language model. For instance, candidates generated using $\mathtt{P_2}$ often involve significant code changes, rarely deleting or simplifying code snippets, but rather creating more advanced solutions. Another observed pattern is that, for the Java dataset and $\mathtt{P_5}$, the generated answers typically contain a `try-catch` block (which is less apparent in the case of JavaScript).

Answer to RQ1: Based on our results, we can conclude that prompts have a major effect on the repair performance of ChatGPT. Among the proposed prompts, we achieved the best results in terms of repair suggestions using $\mathtt{P_1}$ and $\mathtt{P_3}$ in terms of syntactical matches.

The answers generated using prompts $\mathtt{P_3}$ and $\mathtt{P_5}$ often did not include the fix because ChatGPT was unable to generate it. This phenomenon can be observed in cases where significant changes were made during the bug fix, and these prompts essentially delete the modified part, leaving the model with little information about the code's purpose. We also noticed that even small modifications to a prompt can have a significant impact. For example, modifying $\mathtt{P_3}$ to include the number of tokens to be generated (i.e.: `Fill in the missing part in the following code applying X tokens:`)resulted in a significant performance drop. The answers consistently included combinations or reformulated versions of statements such as (1) the missing part cannot be filled with the information provided, (2) in order to complete the code, more context and information about the specific implementation is needed and (3) the information provided is not sufficient for me to understand the context and purpose of the method. The prompt $\mathtt{P_5}$ also often resulted in "undecided" answers. However, here it can be attributed to the fact that the original code is usually syntactically correct in its context, and the bug fix usually only makes sense when observing a larger context or when it is a simple code refinement. Thus, the semantics of the code remain the same.

An interesting insight is that different prompts tend to modify the code in different styles, which suggests the possibility of classifying which prompt fixes which types of bugs, thereby optimizing the repair performance. We observed similar patterns in both Java and JavaScript cases. However, as shown in Figure 2 and Figure 3, ChatGPT suggested significantly more correct fixes for Java. Based on our empirical observations, we hypothesize that this difference is due to the fact that JavaScript developers often use custom object creations (e.g., `{key: val, ...}`) and diverse libraries, while Java follows more standardized conventions with commonly used keywords and methods. Overall, we can conclude that the two languages are distinct and differ greatly in design. Additionally, in the JavaScript dataset, function names are often omitted due to anonymous and arrow functions, whereas in Java, function names are present, which helps the model in understanding the purpose of the function.

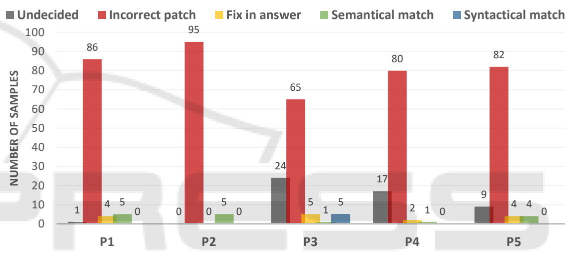


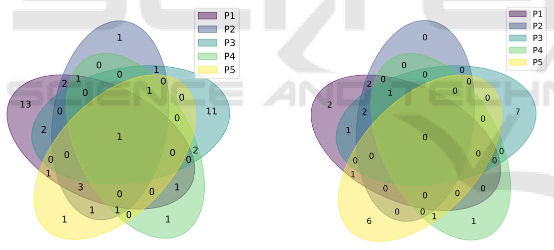
Figure 3: Manually evaluated results of ChatGPT on the JavaScript dataset.

Answer to RQ2: Although we cannot exclude the possibility of dataset bias, based on our empirical evaluation, we observed that ChatGPT tends to generate better repair candidates for Java and less satisfactory ones for JavaScript. Further research in this area is required to determine the exact reasons for this difference in performance.

One might wonder why variable names are not generalized in the used samples, as it is a common practice even in state-of-the-art approaches to reduce vocabulary size (Lu et al., 2021). Without in-depth analysis, we experimented with placeholders but experienced a decrease in performance. It seems that actual variable names and types are beneficial in bugfixing with ChatGPT. Without them, the answers usually included the following observations: (1) TYPE_1 and TYPE_2 are not defined as actual types, and you will need to replace these placeholders with the appropriate types, or (2) METHOD_1 is not implemented, and you will need to provide the implementation. Overall, it appears that language models, such as ChatGPT, contain the most common names and types, so masking them is not beneficial.

We have previously observed that different prompts tend to generate independent fix templates, and this behavior is also observed for Java and JavaScript. For example, `P2` always modified large chunks of code, while `P1` modified only some parts or even left it untouched. Based on these observations, the results in the Venn diagram in Figure 4 are not surprising. The diagram illustrates the distribution of correct fix answers among the used prompts, representing the overlap of fixed bugs using the proposed prompts. In the case of Java, there is only one sample that was fixed by all prompts (a variable name change was necessary, see example 50 in the online appendix), while in the case of JavaScript, there were none. Furthermore, in the Java dataset, ChatGPT repaired 44 different bugs (including answers with the correct fix, semantically identical patches, or syntactically identical ones), while in the JavaScript dataset, it repaired only 24. This also demonstrates that prompts trigger different repair mechanisms, and choosing the right one is a crucial decision.

Answer to RQ3: Since there is insignificant overlap in the fixed bugs and different prompts tend to repair different types of bugs, we did not find particularly easy-to-fix bugs during our experiments. However, some bugs are more likely to be correctly patched by a well-chosen prompt rather than a poorly chosen one.



(a) Java - overall 44 distinct fix (b) JavaScript - overall 24 distinct fix

Figure 4: Distribution of correct fix answers in the used prompts.

4 DISCUSSION

Overall, our results show that choosing the right prompt is a key aspect as it biases the generated fix in many ways. The impact of programming languages is not negligible. It seems that LLMs, such as ChatGPT, tend to generate higher-quality patches for Java (a classic OOP language) and lower-quality patches for JavaScript (an event-driven, functional language). One possible explanation might be that Java is more human-like and its readability is more natural compared to JavaScript. However, to understand the in-depth consequences, further research in the area is needed.

Although the same patterns are apparent in the results, the overlap of fixed bugs is insignificant, and we did not find easy-to-fix bug types in either of the datasets. It is clear that choosing the right prompt holds great importance. On average, ChatGPT was able to propose corrections in approximately 19% of cases, but choosing the wrong input format can drop the performance to as low as 6%. Compared to Transformer models that were fine-tuned to automatically repair programs, this accuracy is not significantly high. For example, on the CodeXGLUE benchmark (Lu et al., 2021), the highest-ranking approach achieved 24% on the small dataset, which is comparable to our data. On the other hand, ChatGPT is surprisingly effective at generating repair candidates, despite not being explicitly trained for that purpose.

As language models continue to improve and evolve, it is possible that prompts that were effective in the past may become less effective or even irrelevant in the future. One approach that can be used to mitigate the impact of model evolution is to periodically re-evaluate the model and prompt selection process to ensure their continued effectiveness. Additionally, we provide the necessary information and code to enable others to replicate our results, even if ChatGPT changes over time.

In addressing our three research questions, we focused on identifying the limitations of using ChatGPT for program repair, which may include several issues such as limited data, difficulty in handling complex programming languages, and potential bias in the model. By answering these questions, we hope to make it clear that the use of ChatGPT for program repair can be beneficial, offering the ability to handle natural language input and the potential to improve developer productivity. Overall, these three questions are interconnected and provide a framework for exploring the potential of using ChatGPT for program repair, identifying the challenges that need to be overcome to realize that potential, and suggesting future research directions.

5 RELATED WORK

ChatGPT has the ability to write and debug computer programs. This capability is not unique to ChatGPT; its predecessor was also capable, although not as flawlessly. Lajko *et al.* (Lajkó et al., 2022; Lajkó et al., 2022) utilized the GPT-2 architecture to automatically repair bugs. While GPT-2 was able to generate some correct fixes, fine-tuning the model did not yield state-of-the-art results. In another work, Prenner *et al.* (Prenner et al., 2022) used Codex for automated program repair, evaluating it on the QuixBugs benchmark (Lin et al., 2017), consisting of 40 bugs in Python and Java. Although they experimented with different prompts, their focus was primarily on code generation from docstrings. Although OpenAI has introduced GPT-4 recently (OpenAI, 2023), the available scientific literature of it is is scarce.

There is no available scientific paper specifically for ChatGPT and most of its use cases are undocumented in scientific literature, we briefly review the use of the GPT family. GPT-2 (Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, 2020) was introduced in 2018, followed by GPT-3 (Brown et al., 2020) in 2020 by OpenAI. They have been applied to various tasks, including poetry, news, and essay writing (Elkins and Chun, 2020; Zhao et al., 2021). The capabilities of GPT have also been explored in the CodeXGLUE benchmark (Lu et al., 2021) for multiple tasks, where CodeGPT achieved an overall score of 71.28 in the code completion task. In a recent work (Ahmad et al., 2021), CodeGPT served as a baseline model for text-to-code and code generation tasks. Another recent work introduced Text2App (Hasan et al., 2021), which enables users to create functional Android applications from natural language specifications.

Now, let's provide a brief summary of the state-of-the-art literature in Automated Program Repair. The Transformer architecture, like GPT, has gained significant attention, and several seminal works have employed it to address the code repair problem using classic supervised training methods (Mastropaolo et al., 2021; Chen et al., 2022; Dinella et al., 2020; Lutellier et al., 2020). In addition to standard training procedures, indirect supervised (Ye et al., 2022) and self-supervised training approaches (Ahmed et al., 2021) have also been explored. Despite breakthroughs in learning-based approaches, standard Generate and Validate (G&V) tools are still widely used due to this day thanks to their availability and configurability (Weimer et al., 2009; Martinez et al., 2017; Kechagia et al., 2022). Defects4J (Just et al., 2014), which consists of 395 Java bugs, is a well-known dataset for various software-related tasks, including APR. ManyBugs (Le Goues et al., 2015) , on the other hand, contains bugs written in C and has been used to evaluate several renowned APR tools (like Genprog (Weimer et al., 2009)). Bugs.jar (Saha et al., 2018) is another notable dataset consisting of 1158 Java bugs and their patches. However, none of these datasets specifically focus on bugs in JavaScript. In the seminal work of Tufano *et al.* (Tufano et al., 2019) created a dataset for Java program repair and evaluated an NMT model on it. This work is also included in the CodeXGLUE benchmark, which encompasses a collection of code intelligence tasks and serves as a platform for model evaluation and comparison.

Recently, numerous Transformer-based models have been introduced, many of which have been evaluated on the CodeXGLUE benchmark. In a recent work (Chen et al., 2022) Chen *et al.* addressed the problem of automatic repair of software vulnerabilities by training a Transformer on a large bug fixing corpus. They concluded that transfer learning works well for repairing security vulnerabilities in C compared to learning on a small dataset. Variants of the Transformer model are also used for code-related tasks, like in (Tan, 2021) where authors propose a grammar-based rule-to-rule model which leverages two encoders modeling both the original token sequence and the grammar rules, enhanced with a new tree-based self-attention. Their proposed approach outperformed the state-of-the-art baselines in terms of generated code accuracy. Another seminal work is DeepDebug (Drain et al., 2021), where the authors used pretrained Transformers to fix bugs automatically. Here Drain *et al.* use the standard transformer architecture with copy-attention. They conducted several experiments including training from scratch, pretraining on either Java or English and using different embeddings. They achieved their best results when the model was pretrained on both English and Java with an additional syntax token prediction.

6 LIMITATIONS AND FUTURE WORK

In this paper, we used a large language model that was trained on a variety of text information, including source code. Despite the fact that ChatGPT was not fine-tuned to repair programs, it is quite effective in this task, albeit with some clear limitations.

Data Leakage - Since the language model used was also trained on source code, we cannot guarantee that the data used was not included in their training set. To address this issue, one would need to know exactly which repositories were included by OpenAI, and that information is not widely available. However, there are mitigating factors: (1) since ChatGPT was trained until a certain period of time, the data used by us is from a different version compared to the data OpenAI might have used; (2) although our dataset satisfies our evaluation criteria, it constitutes only a tiny fraction of the training data used by OpenAI.

Reproducibility - ChatGPT (specifically the model in the correct version) is not openly available, unlike some other LLMs, so reproducibility cannot be ensured. Since the model is exposed via a UI and an API, OpenAI can change the model at any time, even without the user knowing it. To address this limitation, we included input/output prompt samples in the online appendix of this paper.
Prompt Engineering - Although choosing the right prompt is of great importance, in this paper, we did not provide guidance on how to approach the choice of prompts. During our experiments, we observed that a certain prompt triggers a specific repair mechanism, and if these templates could be mapped to bug types, it would guide developers on how to choose the right prompt. We find this research direction to be important and would like to discuss it in more detail in future research.

ChatGPT may be a useful tool for improving software reliability in practice, but one should not blindly trust the code it generates. ChatGPT appears equally confident when generating correct code as it does when generating incorrect code. In future work, we plan to assess the question of automated patch correctness in more detail. We also aim to expand the observed dataset and automate the patch generation process to mitigate dataset bias. Furthermore, further research is needed to determine the exact reasons for performance differences between programming languages in this context.

7 CONCLUSIONS

In this study, the capabilities of ChatGPT were investigated in the field of Automated Program Repair, specifically how it performs when tasked with fixing buggy code. We sampled 200 buggy codes from seminal APR datasets, consisting of 100 Java and 100 JavaScript samples. We designed 5 input prompts for ChatGPT. Our results demonstrate that these prompts have a significant effect on the repair performance, as different prompts trigger different repair mechanisms of the LLM. The overlap of the fixed bugs is negligible. Through manual evaluation of the outputs, we observed that better repair candidates are generated for Java compared to JavaScript. The best prompt for Java generated correct answers in 19% of cases, while for JavaScript, the same prompt yielded a performance of only 4%. In total, 44 distinct bugs were repaired in Java and 24 in JavaScript out of the overall 200 samples and 1000 repair trials. We found that some bugs are more likely to be correctly patched with a well-chosen prompt rather than a poorly chosen one. Therefore, the most important question before starting the repair process is to select the appropriate prompt.

ACKNOWLEDGEMENT

The research presented in this paper was supported in part by the ÚNKP-22-3-SZTE New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund, and by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. The national project TKP2021-NVA-09 also supported this work. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

REFERENCES

(2021). Grammar-Based Patches Generation for Automated Program Repair. *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1300–1305.

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Unified Pre-training for Program Understanding and Generation. pages 2655–2668.

Ahmed, T., Devanbu, P., and Hellendoorn, V. J. (2021). Learning lenient parsing & typing via indirect supervision. *Empirical Software Engineering*, 26(2):1–31.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, I. S. (2020). [GPT-2] Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 1(May):1–7.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T. J., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. *ArXiv*, abs/2005.14165.

ChatGPT: Understanding the ChatGPT AI Chatbot (2023). Chatgpt: Understanding the chatgpt ai chatbot. https://www.eweek.com/big-data-and-analytics/chatgpt/.

Chen, Z., Kommrusch, S. J., and Monperrus, M. (2022). Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering*.

Csuvik, V. and Vidács, L. (2022). Fixjs: A dataset of bug-fixing javascript commits. In *2022 IEEE/ACM 19th*

International Conference on Mining Software Repositories (MSR), pages 712–716.

Dinella, E., Dai, H., Brain, G., Li, Z., Naik, M., Song, L., Tech, G., and Wang, K. (2020). Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. Technical report.

Drain, D., Wu, C., Svyatkovskiy, A., and Sundaresan, N. (2021). Generating bug-fixes using pretrained transformers. *MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021*, pages 1–8.

Elkins, K. and Chun, J. (2020). Can GPT-3 Pass a Writer's Turing Test? *Journal of Cultural Analytics*.

Hasan, M., Mehrab, K. S., Ahmad, W. U., and Shahriyar, R. (2021). Text2App: A Framework for Creating Android Apps from Text Descriptions.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 437–440. Association for Computing Machinery, Inc.

Kechagia, M., Mechtaev, S., Sarro, F., and Harman, M. (2022). Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering*, 48(7):2658–2679.

Lajkó, M., Csuvik, V., and Vidács, L. (2022). Towards javascript program repair with generative pretrained transformer (gpt-2). In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 61–68.

Lajkó, M., Horváth, D., Csuvik, V., and Vidács, L. (2022). Fine-tuning gpt-2 to patch programs, is it worth it? In Gervasi, O., Murgante, B., Misra, S., Rocha, A. M. A. C., and Garau, C., editors, *Computational Science and Its Applications – ICCSA 2022 Workshops*, pages 79–91, Cham. Springer International Publishing.

Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256.

Lin, D., Koppel, J., Chen, A., and Solar-Lezama, A. (2017). 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, page 55–56, New York, NY, USA. Association for Computing Machinery.

Liu, H., Shen, M., Zhu, J., Niu, N., Li, G., and Zhang, L. (2020). Deep Learning Based Program Generation from Requirements Text: Are We There Yet? *IEEE Transactions on Software Engineering*, pages 1–1.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. (2021). CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *undefined*.

Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., and Tan, L. (2020). CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 20:101–114.

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., and Monperrus, M. (2017). Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964.

Mastropaolo, A., Scalabrino, S., Cooper, N., Nader Palacio, D., Poshyvanyk, D., Oliveto, R., and Bavota, G. (2021). Studying the usage of text-to-text transfer transformer to support code-related tasks. *Proceedings - International Conference on Software Engineering*, pages 336–347.

Monperrus, M. (2020). The Living Review on Automated Program Repair. Technical report.

OpenAI (2023). Gpt-4 technical report.

OpenAI ChatGPT (2023a). Openai chatgpt. https://openai.com/blog/chatgpt/.

OpenAI ChatGPT (2023b). Openai chatgpt app. https://chat.openai.com/.

Prenner, J. A., Babii, H., and Robbes, R. (2022). Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. *Proceedings - International Workshop on Automated Program Repair, APR 2022*, pages 69–75.

Saha, R. K., Lyu, Y., Lam, W., Yoshida, H., and Prasad, M. R. (2018). Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. *Proceedings - International Conference on Software Engineering*, pages 10–13.

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., and Poshyvanyk, D. (2019). On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 25–36. IEEE Press.

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 364–374, USA. IEEE Computer Society.

Ye, H., Martinez, M., Luo, X., Zhang, T., and Monperrus, M. (2022). SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics.

Yi, L., Wang, S., and Nguyen, T. N. (2020). Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings - International Conference on Software Engineering*, pages 602–614. IEEE Computer Society.

Zhao, T. Z., Wallace, E., Feng, S., Klein, D., and Singh, S. (2021). Calibrate Before Use: Improving Few-Shot Performance of Language Models.