Better patching using LLM prompting, *via* Self-Consistency

Toufique Ahmed
UC Davis, California, USA
tfahmed@ucdavis.edu

Premkumar Devanbu
UC Davis, California, USA
ptdevanbu@ucdavis.edu

***Abstract*—Large Language models (LLMs) can be induced to solve non-trivial problems with "few-shot" prompts including illustrative *problem-solution* examples. Now if the few-shots also include "chain of thought" ($\mathcal{CoT}$) explanations, which are of the form *problem-explanation-solution*, LLMs will generate a "explained" solution, and perform even better. Recently an exciting, substantially better technique, self-consistency [1] ($\mathcal{S}$-$\mathcal{C}$) has emerged, based on the intuition that there are many plausible explanations for the right solution; when the LLM is sampled repeatedly to generate a pool of explanation-solution pairs, for a given problem, the most frequently occurring *solutions* in the pool (ignoring the *explanations*) tend to be even more likely to be correct!**

Unfortunately, the use of this highly-performant $\mathcal{S}$-$\mathcal{C}$ (or even $\mathcal{CoT}$) approach in software engineering settings is hampered by the *lack of explanations*; most software datasets lack explanations. In this paper, we describe an application of the $\mathcal{S}$-$\mathcal{C}$ approach to program repair, using the *commit log* on the fix as the explanation, *only* in the illustrative few-shots. We achieve state-of-the art results, beating previous approaches to prompting-based program repair, on the MODIT dataset; we also find evidence suggesting that the correct commit messages are helping the LLM learn to produce better patches.

***Index Terms*—LLMs, Self-consistency, Program Repair**

I. Introduction

For more than a decade, language models have found many applications in the field of software engineering. They are based on a simple idea: given a context (or a $prompt$), try to predict the next token (or a missing one); in other words, learn a conditional probability distribution of the form $p(token \mid prompt)$. In neural models, the prompt is a sequence of tokens that is internally represented by a high-dimensional vector, which encodes the parameters of the neural computations. By repeatedly applying this conditional distribution, we can generate sequences of tokens that depend on the prompt and the previously generated tokens (*aka* autoregressive generation).

Modern, instruction-tuned neural language models such as GPT-3 and LLAMA (colloquially known as "LLMs"), can have hundreds of billions of parameters; so when representing a $prompt$ internally, they do so in a very rich and complex space, and then undertake conditional auto-regressive generation starting thereon. The richness of the space for representing prompts allows a wide range of prompt construction, leading to an entirely new field of "prompt engineering"; by guiding LLMs to specific regions of the "context-space" prior, different auto-regressive generation possibilities, conditioned on this prior could ensue, leading to different ways of solving the actual task that one encodes in a prior. Approaches include few-shot learning [2], and chain-of-thought [3]. "Few shot" ($\mathcal{FS}$) learning amounts to providing examples (input/output *pairs*) illustrating the task within the prompt, and then asking for the output for a target input; "chain-of-thought" ($\mathcal{CoT}$) amounts to providing *reasons* connecting input and output pairs, thus input-reason-output *triples*. $\mathcal{FS}$ and $\mathcal{CoT}$ are complementary. All these techniques work in software engineering tasks.

Since these generative possibilities amount to sampling sequentially from the next-token distribution that the LLM has learned, different sequences could be generated. Normally, one follows a *greedy sampling* approach, taking the most likely token at each stage of the auto-regressive generation.

In a recent paper [1] Wang *et al* proposed a different approach, to selecting an output sequence from an auto-regressively trained model, called "self-consistency" ($\mathcal{S}$-$\mathcal{C}$). With $\mathcal{S}$-$\mathcal{C}$, the idea is that the model is prompted (using $\mathcal{CoT}$, perhaps with $\mathcal{FS}$) to produce *first* an *explanation*, and then an *answer* . . . but for $\mathcal{S}$-$\mathcal{C}$, the LLM is sampled repeatedly, using a "high temperature"[1]. This repeated sampling can be thought of as a way to model to generate completions (first *explanations*, then *answers*) from "different perspectives". From these varied "perspectives", $\mathcal{S}$-$\mathcal{C}$ simply chooses the most frequent *answer* (thus ignoring, or "marginalizing over" the explanations). $\mathcal{S}$-$\mathcal{C}$ is remarkably effective, showing significant improvements on many tasks in Natural language. In this paper, we explore application of this idea to program repair.

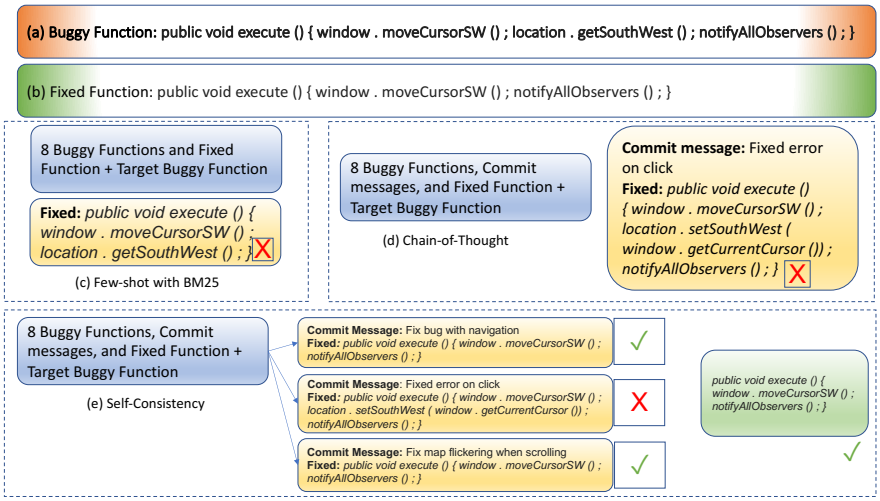
Our primary contributions are:

1) We find evidence suggesting that $\mathcal{S}$-$\mathcal{C}$, with commit-log message as the reason, improves performance on the code-repair task.
2) Our data suggests that using the correct commit log messages within the few-shots actually does help produce sigificantly better answers, with $\mathcal{S}$-$\mathcal{C}$; random commit messages do not.

This material is based upon work supported by the National Science Foundation under Grant NSF CCF (SHF-MEDIUM) No. 2107592. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[1]Temperature $t, t \in [0, 1]$ here refers to the likelihood of choosing a next token that's not necessarily the most likely following token as per the model. $t = 0$ is just greedy selection.

Fig. 1. Steps followed for $\mathcal{FS}$,$\mathcal{CoT}$, and $\mathcal{S}$-$\mathcal{C}$ in program repair.



II. Background & Related Work

A. Few-shot Learning

$\mathcal{FS}$ learning is very useful with (LLMs) [2], [4]. With $\mathcal{FS}$ learning, *e.g.,* if we desire to translate English to German, we assemble a few English-input, German-output pairs into a $prompt$. Now if a final English sentence is added to the prompt, the model completes the prompt with German translation theoreof. This trick works for a range of tasks; remarkably, the model does this, *without requiring any weight updates*. While $\mathcal{FS}$ learning originated in NLP applications, it also works for a wide range of software engineering tasks, including code summarization [5], code repair [6], assertion generation [6], code mutation [7], test oracle generation from natural language documentation [7], and test case generation [7]. Notably, using $\mathcal{FS}$ on these tasks appears to improve performance compared to previous state-of-the-art models. $\mathcal{FS}$ learning is felicitously helpful when even just a few labeled/curated examples are available; but for many SE tasks, data is often mined from repositories, and is more abundant. Can this more abundant data be leveraged in a few $\mathcal{FS}$ setting?

Nashid et al. [6] used an IR-based approach (BM25 [8]) to select specific few-shot samples, which improved $\mathcal{FS}$ performance for both code repair and assertion generation. Ahmed et al. found that this approach also improves code summarization performance [9]. Fig. 1-(c) illustrates the input-output of the BM25-based few-shot learning method for program repair task, where eight (can be more or less depending on the context length) pairs of buggy-fixed function examples were retrieved using the BM25 algorithm and concatenated with the buggy version of the target function. Consequently, the fixed version of the function was retrieved from the model's output.

B. Chain-of-Thought ($\mathcal{CoT}$)

Wei et al. [3] found that forcing the generation of intermediate reasoning steps (*viz.,* Chain of Thought ($\mathcal{CoT}$)), substantially improves LLM performance on complex reasoning tasks. As with $\mathcal{FS}$ learning, for $\mathcal{CoT}$, we prompt the model with a few input-output pair "shots"; but now, each shot is augmented with an intervening *reasoning path*: each shot is now an ordered triplet of $\langle input, reasoning\text{-}path, output \rangle$. A prompt includes several triplets, followed by a target $input$; the model then generates the $reasoning\text{-}path$, and the $output$. This approach substantially improves task performance in several settings, including commonsense, and symbolic reasoning.

But this approach requires atleast *some* few-shots with $reasoning\text{-}path$s. In general, SE datasets (for repair, summarization, code-retrieval *etc*) have abundant $\langle input, output \rangle$ pairs, but $reasoning\text{-}path$s are rare. Creating these paths, for few-shotting, after-the-fact, can be a challenge.

We posit that *a summary or commit log message could serve as a reasoning path*. Fig. 1-(d) illustrates the model output when the model is prompted to generate the reason/commit message prior to generating the fixed function. The MODIT [10] work used commit messages in program repair tasks. However, in MODIT, commit messages *associated with the* **target** $input$ were actually inserted in the prompt[2]. As we shall see next, for the $\mathcal{S}$-$\mathcal{C}$ approach, the reasoning path (commit message) associated with the target *must also be* generated, in order to allow model some additional randomness to subsequently generate a range of different possible outputs for the target input.

C. Self-Consistency ($\mathcal{S}$-$\mathcal{C}$)

Wang et al. [1] introduced self-consistency-based generation, which improves over the naive greedy decoding approach used in chain-of-thought prompting. $\mathcal{S}$-$\mathcal{C}$ starts with a prompt with few-shot $\mathcal{CoT}$ triples ending with a target $input$, thus prompting the generation of a $reasoning\text{-}path$ and an $output$. So far, similar to conventional $\mathcal{CoT}$.

But now, $\mathcal{S}$-$\mathcal{C}$ posits higher-temperature generation, thus sampling a *collection* of distinct reasoning paths and outputs, rather than greedily sampling (with 'Temperature 0') a single reasoning path and output. From this collection, they select

[2]The MODIT approach arguably is not leaking test-data, since it's very plausible that a developer fixing a bug knows what needs to happen, and can thus write commit log, before they actually code-up the change.

TABLE I
PERFORMANCE OF CHAIN-OF-THOUGHT AND SELF-CONSISTENCY IN PROGRAM REPAIR TASK

Dataset	Greedy	$\mathcal{CoT}$	$\mathcal{S}$-C	Relative Gain over Greedy	p-value	Greedy + BM25	$\mathcal{CoT}$ + BM25	$\mathcal{S}$-C + BM25	Relative Gain over Greedy + BM25	p-value
$B2F_s$	9.50%	10.00%	13.50%	+42.10%	< 0.01	29.00%	29.00%	31.80%	+9.65%	< 0.01
$B2F_m$	11.20%	10.40%	15.50%	+38.39%	< 0.01	19.10%	20.20%	21.60%	+13.08%	< 0.01

the most frequently occurring output (ignoring the reasoning paths). $\mathcal{S}$-C builds upon the intuition that the same correct answer to a complex problem is often reached *via* several distinct reasoning approaches. The $\mathcal{S}$-C expansion of $\mathcal{CoT}$ prompting is remarkably effective. In this study, we generate upto 50 samples for each input sample, and the final model output is determined through majority voting. Fig. 1-(e) showcases an example of self-consistency with three samples.

It's important to note here that $\mathcal{S}$-C requires the generation of a reasoning-path *after* the target input in the prompt, and *before* the required output. To prompt an LLM to do this, we do require the same kind of triplet as used in $\mathcal{CoT}$. Unfortunately, in SE, while datasets for tasks include inputs and outputs, $reasoning\text{-}path$ element is almost never available. Our idea here was to use commit-log messages as the $reasoning\text{-}path$, and then apply $\mathcal{S}$-C to see if new SOTA performance can be achieved for program-repair, on the MODIT dataset.

D. Research Questions (RQs)

Our first question considers the value of the $\mathcal{S}$-C approach for program repair tasks. We consider the setting where the model's input is the complete buggy function *without* bug localization, and the output is the complete fixed function. Currently, BM25-based few-shot retrieval has reached a high-water mark [6] in this setting. We study if BM25, when combined with self-consistency, delivers even better performance.

RQ 1. Does $\mathcal{S}$-C improve program repair performance?

When using $\mathcal{S}$-C, one has to sample a number of reasoning paths, over which to find consistent outputs. More paths might lead to better consistency; but computational costs scale also linearly with more paths, so it would be good find the balance.

RQ 2. How does performance vary with the number of generated reasoning paths?

Finally: our few-shot examples include the commit message, actually associated with the bug-fix commit, as the "reasoning path", to push the model to generate a good reasoning-path for the target input. But are these *actual* commit messages matter? Is the model actually learning from these reasoning paths in the few-shots? To study this question, we prepared few-shot samples where the commit message was *not* the one associated with the fix, but *some other random* commit message sourced from the data. If commit messages are indeed useful "reasoning paths", such random "reasoning paths" *should* confuse the model and diminish performance.

RQ 3. How does the performance change with use of random commit messages instead of the original ones?

III. METHODOLOGY

A. Dataset

We use the MODIT dataset, which includes two subsets ($B2F_s$, which has smaller sequences, and $B2F_m$) [11]. MODIT comprises bug-fix commits (including *commit-logs*), from GitHub. Variants of these datasets have been used in evaluating NatGen [12], CodeT5 [13], and also in the CodeXGLUE [14] benchmarks. Other work [6] used the TFix [15] dataset, which is classified into the 52 error types flagged by ESLint [3]. The MODIT dataset, includes commit messages written by developers; we use these commit-messages as the Chain-of-thought (which is required if one seeks to apply $\mathcal{S}$-C to a task). In our experiment, we use the training partition as the sample pool from which BM25 retrieves relevant few-shot samples. From the test partition, we randomly sampled 1000 examples for our tests, to gain statistical power while also dealing with OpenAI's rate limitations & associated costs.

Program repair is a popular topic; conducting a comprehensive comparison of all the models [16]–[20] and datasets [15], [21], [22] is beyond the scope of this paper; we're interested to see if commit log messages can be used as chain-of-thought for applying self-consistency.

B. Model

One can access models of varying sizes, that are trained using code and related NL descriptions. Models like Codex [4], PaML [23] PolyCoder [24], and CodeGen [25] have gained significant popularity. Our primary focus is on the Code-DaVinci-002 model, which has 175 billion parameters and demonstrate exceptional performance in code-related tasks. We access this model *via* OpenAI API.

C. Proposed Approach, Baseline & evaluation criteria

We base-line $\mathcal{S}$-C, on the program repair task, over the state-of-the-art Nashid *et al* [6], which uses BM25-retrieved few-shots, with greedy decoding. We tried several $\mathcal{FS}$ decoding techniques to generate correct patches: greedy decoding (temperature 0), $\mathcal{CoT}$, and $\mathcal{S}$-C. Greedy-decoding just sequentially selects the most probable output. $\mathcal{CoT}$ also decodes greedily: but it uses triplet-shots in the few-shot prompt to induce the model to follow-up the target input with a reasoning-path before generating the output. For $\mathcal{S}$-C, we use a temperature of 0.7 to configure the model to generate more diverse reasoning-paths & outputs for the target input. We sampled up to 50 sequences for each of the 1000 target test cases during the experiment. However, when reporting the results, we focused on the first 30 samples: we found that the performance tends

[3]See https://palantir.github.io/tslint/

Fig. 2. Number of $\mathcal{S}$-C samples vs. Acuuracy

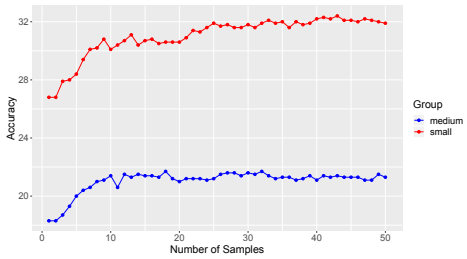


TABLE II
IMPACT OF USING RANDOM COMMIT MESSAGES.

Dataset	Acc.	Comparing with baseline	
		Gain	p-value
$B2F_s$	30.00%	+3.44%	0.30 *(not Sig.)*
$B2F_m$	19.90%	+4.18%	0.30 *(not Sig.)*

to plateau after the initial 10 samples (Section IV-B). As with [6], BM25 was used to find relevant few-shots. We repeated all three approaches (greedy, $\mathcal{C}o\mathcal{T}$ and $\mathcal{S}$-C), using samples chosen by BM25. Using the SOTA approach [6] (BM25) as a baseline, we specifically examine the benefits of using $\mathcal{S}$-C in conjunction with BM25. We used the exact match as evaluation metric.

IV. RESULT

A. *RQ1: Performance of the proposed approached*

Table I shows the performance (top-1 exact match with the fix) of $\mathcal{C}o\mathcal{T}$ and $\mathcal{S}$-C in the program repair task. Using BM25 to select few-shots works best. We also show results for the traditional fixed 8-sample $\mathcal{FS}$ learning: $\mathcal{S}$-C improves over greedy decoding (*sans* BM25) by 42.10% and 38.39% (relatively). However, when few-shots are selected using BM25, the gains are lower, 9.65% and 13.08% respectively. To determine statistical significance, we performed a McNemar test [26] (a non-parametric test used to analyze matched nominal data). For each pair of settings above, self-consistency improves ($p < 0.01$) over greedy decoding. We also observe that the using $\mathcal{C}o\mathcal{T}$ sometimes provides improvement, albeit not sufficient to be statistically significant.

B. *RQ2:How many samples are needed for $\mathcal{S}$-C?*

To manage costs of repeated sampling, we study how performance changes with number of samples. Fig. 2 suggests that the performance consistently improves for the first 10 samples for both datasets. However, for $B2F_m$, we saw no significant benefits beyond 10 samples. On the other hand, for $B2F_s$, the performance improves with an increasing number of samples, though the benefits taper off. To ensure generality in our reporting, we consider 30 samples as the chosen number for our results and the complete processing takes less than 5 seconds for each sample (ignoring rate limitations).

C. *RQ3: Random commit message vs. original ones*

We tried pairing the buggy program with random commit messages instead of the commit messages in $\mathcal{FS}$ samples. We saw lower improvement of 3.44% and 4.18% over our baseline approach; moreover, statistical significance was not achieved (Table II). However, using the original or correct commit messages resulted in statistically significant improvement: this suggests that the LLM is learning better reasoning paths from the original commit messages, for the task of program repair.

V. DISCUSSION

Before LLMs, encoder-decoder models were commonly employed to fix buggy programs. These models were fine-tuned using the entire training data and exhibited satisfactory performance on this specific task. A recent model called NatGen [12] scored an accuracy of 23.43% and 14.93% on $B2F_s$ and $B2F_m$, respectively. These numbers are lower than our scores of 31.80% and 21.60% accuracy; note that Natgen *included* the commit log message for the test example, we don't; we *generate* it, but then discard it, and retain just the fix code. It is important to note that our performance wasn't measured on the full test set, rather on just a sample; Nevertheless, since we randomly chose a large (1,000) sample for our experiments, one can reasonably expect that our findings are robust (note $p < 0.01$ as per McNemar test for our main finding).

The commit messages in MODIT are sometimes uninformative, and limited to "Bug Fixed". With commit messages of better quality, the model's performance with $\mathcal{C}o\mathcal{T}$ and $\mathcal{S}$-C could potentially improve. It is possible to generate informative commit messages with the assistance of other models, but further research is required. Several studies have reported higher performance compared to our approach [10], [27]. However, we note that these works assume that the location of the bug is known, and they generate fix "snippets" instead of complete fixed functions. This assumption may limit use if the fault location is not known; we, however, pass in the whole buggy function to the model [10], without fault localization, and require the entire fixed function as output.

VI. CONCLUSION

This paper explores the idea of using self-consistency for defect repair, specifically using the commit-log message as the "reasoning path" just within the few-shot illustrative examples (but no commit-log message for the test example). We find that this approach provides improved performance; furthermore, we find that generating a sample-pool of around 30 explanation-answer pairs, and choosing the most frequent answer, works well. We also find that evidence suggesting that the LLM is learning from the *actual, correct* commit-log message in the few-shots: using random commit-log messages doesn't provide any significant improvement over prior approaches. For our data and Script, please access https://doi.org/10.5281/zenodo.7968641.

REFERENCES

[1] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[5] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[6] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[7] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code," *arXiv preprint arXiv:2206.01335*, 2022.

[8] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[9] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving few-shot prompts with relevant static analysis products," *arXiv preprint arXiv:2304.06815*, 2023.

[10] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[11] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[12] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30.

[13] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[14] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[15] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[16] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[17] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[18] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. 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*, 2020, pp. 101–114.

[19] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1506–1518.

[20] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[21] R. Just, D. Jalali, and M. 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*, 2014, pp. 437–440.

[22] D. Lin, J. Koppel, A. Chen, and A. 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*, 2017, pp. 55–56.

[23] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[25] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[26] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[27] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.