Automated Program Repair in the Era of Large Pre-trained Language Models

Chunqiu Steven Xia
University of Illinois
Urbana-Champaign
chunqiu2@illinois.edu

Yuxiang Wei
University of Illinois
Urbana-Champaign
ywei40@illinois.edu

Lingming Zhang
University of Illinois
Urbana-Champaign
lingming@illinois.edu

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00129

Abstract—**Automated Program Repair (APR) aims to help developers automatically patch software bugs. However, current state-of-the-art traditional and learning-based APR techniques face the problem of limited patch variety, failing to fix complicated bugs. This is mainly due to the reliance on bug-fixing datasets to craft fix templates (traditional) or directly predict potential patches (learning-based). Large Pre-Trained Language Models (LLMs), trained using billions of text/code tokens, can potentially help avoid this issue. Very recently, researchers have directly leveraged LLMs for APR without relying on any bug-fixing datasets. Meanwhile, such existing work either failed to include state-of-the-art LLMs or was not evaluated on realistic datasets. Thus, the true power of modern LLMs on the important APR problem is yet to be revealed.**

In this work, we perform the first extensive study on directly applying LLMs for APR. We select 9 recent state-of-the-art LLMs, including both generative and infilling models, ranging from 125M to 20B in size. We designed 3 different repair settings to evaluate the different ways we can use LLMs to generate patches: 1) generate the entire patch function, 2) fill in a chunk of code given the prefix and suffix 3) output a single line fix. We apply the LLMs under these repair settings on 5 datasets across 3 different languages and compare different LLMs in the number of bugs fixed, generation speed and compilation rate. We also compare the LLMs against recent state-of-the-art APR tools. Our study demonstrates that directly applying state-of-the-art LLMs can already substantially outperform all existing APR techniques on all our datasets. Among the studied LLMs, the scaling effect exists for APR where larger models tend to achieve better performance. Also, we show for the first time that suffix code after the buggy line (adopted in infilling-style APR) is important in not only generating more fixes but more patches with higher compilation rate. Besides patch generation, the LLMs consider correct patches to be more *natural* than other ones, and can even be leveraged for effective patch ranking or patch correctness checking. Lastly, we show that LLM-based APR can be further substantially boosted via: 1) increasing the sample size, and 2) incorporating fix template information.

I. Introduction

As software programs and systems become more and more ubiquitous in everyday life, so do software bugs. Due to the wide-ranging adoption of software systems in fields from healthcare [1] to transportation [2], these bugs can potentially cause dangerous safety issues [3] and financial losses [4]. As such, developers often need to spend a significant amount of time and effort to fix software bugs [5]. In order to help developers reduce this manual effort, Automated Program Repair (APR) tools have been built to automatically generate potential patches given the original buggy program [6].

Among traditional APR techniques [7]–[18], template-based APR has been widely recognized as the state of the art [19], [20]. These techniques leverage fix templates, often designed by human experts, to fix specific types of bugs in the source code. As a result, these APR tools are constrained by the underlying fix templates in the types of bugs that can be fixed. To combat this, researchers have proposed learning-based APR tools [21]–[24], which typically model program repair as a Neural Machine Translation (NMT) problem [25], where the goal is to translate a buggy program into a fixed program. The core component of these learning-based APR tools is an encoder and decoder pair, where the model aims to capture the buggy context via the encoder and then autoregressively generate the patch using the decoder. As such, these learning-based APR tools require supervised training datasets containing pairs of buggy and patched code, usually obtained by mining historical bug fixes from open-source repositories. While learning-based APR tools have shown improvements in both the number and variety of bugs that can be fixed [21], [22], they are still restricted by their training data which may contain unrelated commits and only contain limited bug-fix types, which may not generalize to unseen bug types [26].

Recent developments in building Large Pre-Trained Language Models (LLMs) offer an alternative solution that can be applied for program repair without relying on historical bug fixes. While LLMs are usually general-purpose tools for NLP tasks (e.g., GPT3 [27]), they have also been used for programming languages by finetuning on code (e.g., Codex [28] and ChatGPT [29]). Unlike the specifically designed learning-based APR models, LLMs are trained in an unsupervised fashion using up to billions of text/code tokens and can be used in a variety of code tasks. Recently, AlphaRepair [26] proposes to leverage CodeBERT [30], a large code model pre-trained on millions of code snippets, directly for APR. The key insight from AlphaRepair is instead of learning transformations to go from buggy code to fixed code, we can directly use the model to predict what the correct code should look like given its surrounding context (including both prefix and suffix), i.e., *infilling-style* APR. Using this idea, AlphaRepair demonstrated state-of-the-art repair results without finetuning on bug fixing dataset. While AlphaRepair has shown improvements over

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00129

previous learning-based APR, the model (125M parameters) it uses is far smaller than the current state-of-the-art LLMs (Codex: 12B parameters and GPT-3: 175B parameters). Beside AlphaRepair, researchers have also directly leveraged Codex for *generative* APR [31], [32], i.e., generating the fixes based on the context before bugs (i.e., prefix only). However, these studies mostly focus on Codex and are only evaluated on a small dataset with 40 bugs on simple programming tasks.

Current state-of-the-art LLMs [28], [33] have also included evaluation for code related tasks such as code completion [28], docstring generation [34] and variable/type prediction [34]. However, these evaluations still mainly focus on NLP metrics such as BLEU score [35] which do not accurately measure the functional or semantic correctness of the generated code. Furthermore, the datasets consist of hand-curated code problems which do not accurately reflect the type of projects developers work on in the real world.

Our Work. We present the first extensive evaluation of recent LLMs for fixing real-world projects. We designed 3 different APR experimental settings: 1) complete function generation 2) correct code infilling and 3) single line generation to showcase the different ways LLMs can be applied for APR. In our study, we include both popular types of LLM architectures (generative and infilling models) to show the advantages and flaws of using each type for APR. We include models with a wide range of different parameter sizes, spanning from 125 million to 20 billion. We evaluate not only the improvement in repair effectiveness but also the trade-off with respect to speed when increasing the model size. In total, we use 5 different repair datasets containing real open-source bugs and developer written tests across 3 programming languages to evaluate APR under realistic settings. Compared with existing applications of LLMs for APR [26], [31], [32], our study is the first to include state-of-the-art LLMs for both infilling-style and generative APR on various datasets and programming languages. To summarize, this paper makes the following contributions.

- ⋆ **Dimension.** This paper bridges the gap between the recent advances in LLMs and a crucial software engineering problem – APR. This paper not only demonstrates the potential and future for directly leveraging LLMs for solving the important APR problem, but also provides a realistic evaluation scenario for the recent LLMs, which were mainly evaluated on simple/synthetic coding problems rather than real-world systems as studied in the APR area.
- ⋆ **Study.** We conduct extensive evaluations using 9 different recent LLMs on 5 different repair datasets across 3 different programming languages (Java, Python, and C). We compare the LLMs against each other using the 3 repair settings we designed. Using the popular repair datasets, we further compare the LLMs with state-of-the-art APR tools.
- ⋆ **Practical Guidelines.** Our study shows for the first time that directly applying state-of-the-art LLMs can already substantially outperform all existing APR tools on the widely studied Defects4J 1.2 dataset (and other ones), e.g., Codex can fix 32 more bugs than the existing best APR technique. Among the studied LLMs, the scaling effect exists for APR where larger models tend to deliver stronger APR results. Also, we show for the first time that suffix code after the buggy line (adopted in infilling-style APR) is important in not only generating more fixes but more patches with higher compilation rate. Besides patch generation, the LLMs consider correct patches to be more *natural* than other ones, and can even be used for effective patch ranking or correctness checking. Lastly, we show that LLM-based APR can be further substantially improved via: 1) increasing the sample size, and 2) incorporating fix template information.

II. Background and Related Work

A. Large Pre-Trained Language Model

Large Pre-Trained Language Models (LLMs) have become ubiquitous in the domain of NLP, achieving impressive performance in many tasks such as machine translation [25], text summarization [36] and classification [37]. LLMs follow the Transformer architecture [38] – an encoder to capture input representation and a decoder to generate output tokens. These LLMs are first pre-trained in an unsupervised manner, on large amounts of text data and then finetuned for downstream tasks. However, certain tasks may not have an abundance of finetuned data available. As such, researchers have evaluated the ability for LLMs to perform on downstream tasks without finetuning. This is achieved via prompt engineering [39] – providing the model with natural language descriptions and demonstrations of the task it is trying to solve before giving the model the target input. This works by leveraging the general-purpose setup of LLMs where the unsupervised pretraining dataset already encompasses many domains of problems/tasks. Using this idea and the exponential growth in LLM size [40], impressive performance in many tasks can be achieved even without any finetuning [27].

LLMs can be classified into encoder-only, decoder-only and encoder-decoder models based on their architectures. Encoder-only models (such as BERT [41]) contain only the *encoder* component of a Transformer. They are typically designed to learn data representations and are trained using the Masked Language Modeling (MLM) objective – a small percentage (e.g., 15%) of tokens in the training data will be replaced by masked tokens, and then the models are trained to predict the original values of the masked tokens based on the bidirectional contexts. Decoder-only models (such as GPT-3 [27] and GPT-Neo [42]) are large generative models that use the *decoder* to predict the next token output given all previous tokens (i.e., left context or prefix only). To combine the usage of both encoder and decoder, encoder-decoder models (such as T5 [43] and BART [44]) have also been proposed for sequence-to-sequence tasks where the training objective aims to recover the correct output sequence given the original input (e.g., corrupted to uncorrupted). One such training objective is span prediction tasks, where random spans (multiple tokens) are replaced with artificial span tokens and the model is tasked with recovering the original tokens. For inferencing, one can use the encoder-decoder models to infill text by also adding the artificial span token in place. Recently, researchers have also combined MLM with generative models to perform both bidirectional and autoregressive text generation or infilling [45]. In our APR scenario, all types of LLMs can potentially be leveraged for generative or infilling-style APR, and we select 9 state-of-the-art LLMs for our study (detailed in Section III-A).

B. Automated Program Repair

Automated Program Repair (APR) tools are used to generate patched code given the original code and the corresponding buggy location. Each patch generated by the APR tool is validated against the test suite. *Plausible patches* are ones which pass the entire suite. *Correct patches* are plausible patches which correctly fix the underlying bug.

Traditional APR tools can be classified as heuristic-based [7]–[9], constraint-based [10]–[12] and template-based [13]–[16], [19]. Traditionally, template-based APR tools achieve the best performance, where each template is hand-crafted by human experts designed to provide a fix for a specific type of bug. However, these template-based APR tools can only fix the bug types that are part of the templates. As a result, researchers employed learning-based APR tools to generate more expressive patches. Learning-based APR tools such as Recoder [21], RewardRepair [23], and CURE [22] are based on NMT techniques [25] which require specific bug fixing data to train the NMT model to generate a fix line given the buggy line. Due to this reliance on the bug-fixing data, these learning-based tools are still limited in terms of the type of fixes it can apply. Recent work of AlphaRepair [26] addresses this by performing APR under a zero-shot setting by directly using the CodeBERT model for repair. AlphaRepair fills the original buggy line with masked tokens and uses CodeBERT to replace the masked tokens with correct code tokens to generate repair, i.e., *infilling-style* (also called *cloze-style*) APR. While AlphaRepair is able to achieve state-of-the-art results, CodeBERT is considerably smaller than the newest LLMs. Additionally, AlphaRepair is designed for the repair setting where the buggy line location is known (e.g., computed by fault localization techniques [46]).

Recent work [31], [32] has also looked into directly applying LLMs for APR. Prenner et al. [32] conducted a small-scale evaluation for the Codex model on a simple dataset containing both Java and Python versions of buggy algorithm implementations. Codex is given the buggy function and by using prompt engineering, are then asked to generate a complete fixed function. The results show that Codex is competitive with state-of-the-art learning-based APR tools in Python but worse in Java. In contrast, we show that by using our repair settings, LLMs are able to outperform state-of-the-art APR tools on both Java and Python. Kolak et al. [31] also used Codex along with 2 smaller LLMs and evaluated their ability to generate the correct patch line when given the code prefix on the same dataset as the previous work [32]. The evaluation demonstrated the scaling effect of LLMs where the repair results can be improved by using larger models. Interestingly, the study leverages sum entropy for patch ranking while AlphaRepair leverages mean entropy (i.e., both favors more *natural* [47] patches). Thus, we also perform a study of leveraging various recent LLMs for computing both entropies for patch ranking on real-world systems. In addition, to the best of our knowledge, we are the first to study LLMs or entropies for patch correctness checking (i.e., distinguishing correct patches from plausible ones).

TABLE I: Studied LLMs

Model	#Parameters	Training Dataset	Type
GPT-Neo	125M/1.3B/2.7B	The Pile	Generative
GPT-J	6.7B	The Pile	Generative
GPT-NeoX	20B	The Pile	Generative
Codex	12B	N.R.	Generative & Infilling
CodeT5	220M	CodeSearchNet & BigQuery	Infilling
InCoder	1.3B/6.7B	N.R.	Infilling

Overall, the 2 prior studies [31], [32] are done on a small dataset with synthetic bugs using only a small number of LLMs. Moreover, the input and repair setting being used in the studies are also limited, e.g., only considered *generative* APR. In this paper, we present an extensive study of applying various state-of-the-art LLMs for both infilling-style and generative APR on diverse repair datasets across programming languages.

III. Approach

In this section we describe the LLMs selected for evaluation and introduce 3 different APR generation settings we use to evaluate each LLM. These settings are designed to showcase the different practical ways we can directly use LLMs for APR and highlight advantages and differences of the studied LLM types. Also, we detail the patch ranking strategy of using entropy to prioritize patches that are more likely to be correct.

A. Models

We begin by describing the different LLMs we use for evaluation. Our selection process starts with the list of popular models hosted on the Hugging Face [48] – an open-source platform to host and deploy large models. We sort the list of models based on popularity (#downloads this month) and select the LLMs which contain code as training data. Furthermore, we also pick models from different organizations and types (described below) to obtain a diverse set of models. Along with the open-source models, we also use the closed-source Codex model [28] (accessible only via API) since it has shown to achieve impressive performance on code related tasks. In total, we use 9 different LLMs for our experiment.

Our chosen LLMs range from 125M to 20B in parameter size. Table I presents the LLM overview. Column **Model** is the model name, **#Parameters** presents the number of model parameters, **Training Dataset** indicates the dataset used for pre-training (N.R. is not released), and **Type** refers to the type of APR the model can perform (infilling or generative).

1) Generative Models:

- **GPT-Neo [42], GPT-J [49], GPT-NeoX [50]** All three models are open-source implementations of the GPT-3 transformer architecture [27]. In our experiments, we use GPT-Neo models with 125M, 1.3B and 2.7B parameters. GPT-J and GPT-NeoX are even larger models with 6.7B and 20B parameters. These models were trained on The Pile [51], an 800GB dataset combining 22 diverse text-based datasets with 7.6% containing open-source Github code.

- **Codex [28]** A 12B parameter GPT-3 based model designed for code generation. Codex is initialized with GPT-3 weights trained on natural language corpus and then finetuned on a large corpus of 159GB code files.

2) Infilling Models:

- **CodeT5 [52]** A 220M parameter model based on T5 [43] architecture designed for code related tasks. CodeT5 is trained using span prediction objective on 8.35 million functions across 8 different programming languages by combining CodeSearchNet [53] with C/C# dataset from BigQuery [54].
- **INCODER [33]** A model designed for code infilling by adopting a causal masking objective [45]. INCODER is trained on both open-source Github/GitLab code (159 GB) and StackOverFlow questions and answers (57 GB). We use both the 1.3B and 6.7B parameter version.
- **Codex** In addition to using Codex as a generative model, we use the recently added suffix feature [55] to perform code infilling. Since Codex is not open-sourced, we do not know how the model performs the infilling.

B. LLM-based Patch Generation

In our study, we designed three settings for APR:

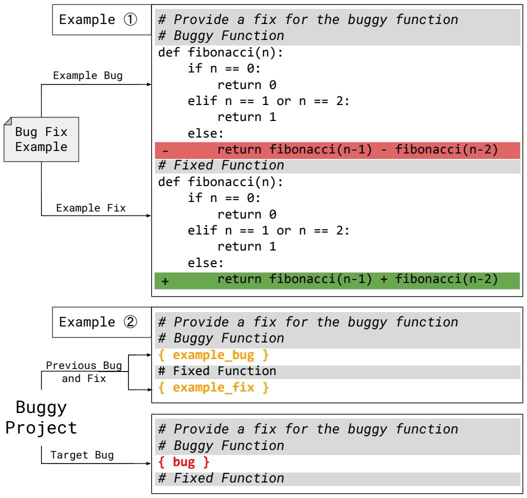
1) Complete function generation – the input is a buggy function and the goal is to output the patched function.

2) Correct code infilling – the buggy location is known and the goal is to generate the correct replacement code given the prefix and suffix of the buggy function.

3) Single line generation – the bug location is provided and the bug is fixed by a single line change. Single line generation uses a subset of bugs in correct code infilling. We separate this case since many fault-localization techniques provides a ranking in the granularity of individual code lines [46], [56]. More importantly, both infilling and generative LLMs can be applied for this setting, enabling direct comparison of the two. We now describe the different inputs for each setting.

1) Complete function generation: For this setting, the initial input is the original buggy function. We aim to use a generative model to autoregressively generate the entire patched version of the buggy function. However, naively feeding the LLMs the buggy function will not work since each LLM is not pre-trained for APR (i.e., they do not know that the goal is to generate a patched function). Therefore, to facilitate the direct usage of LLMs for APR, we use specific prompts to enable the models to perform few-shot learning. This allows the LLMs to recognize the task and generate a patched function by completing the input provided. We note here that the task of complete function generation makes no assumption of 1) the location of the bug and 2) the type of bug or fix required. Therefore, the LLM needs to figure out *why* the function is buggy and provide a patch to fix the bug.

Figure 1 shows the input which is made up of two example bug fixes (one crafted by us and one from the same project/dataset the target bug is from) in order to demonstrate the task and the expected format of the output. To start off, we follow the prior study [32] and begin with a description of the task: `# Provide a fix for the buggy function`. This describes in natural language the task we want the LLM to perform. This is a Python example and we use the Python comment format of `#` as a prefix for this description (we use other comment prefixes depending on the language of the buggy code). We then provide an example bug and patch pair. In Figure 1, this example is a function which computes the Fibonacci number. We prefix the example buggy and fixed function with `# Buggy Function` and `# Fixed Function` to provide additional context for the model. For our second example, we follow the same prompting style and pick a buggy and patched function pair from the same project that the bug is from. This way we can provide the model with some examples of the coding style used in the project. Finally, we finish the prompt by adding the bug we want to fix.

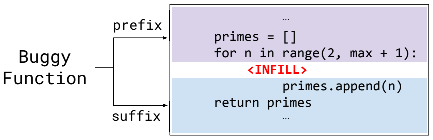


```
# Provide a fix for the buggy function
# Buggy Function
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
-       return fibonacci(n-1) - fibonacci(n-2)
# Fixed Function
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
+       return fibonacci(n-1) + fibonacci(n-2)
```

```
# Provide a fix for the buggy function
# Buggy Function
{ example_bug }
# Fixed Function
{ example_fix }
```

```
# Provide a fix for the buggy function
# Buggy Function
{ bug }
# Fixed Function
```

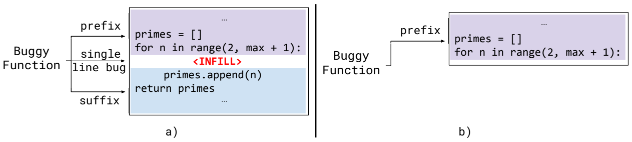
Fig. 1: APR input for complete function generation



```
                …
    primes = []
    for n in range(2, max + 1):
            <INFILL>
            primes.append(n)
    return primes
                …
```

Fig. 2: APR input for correct code infilling

2) Correct code infilling: Unlike complete function generation, where the buggy location within the function is not known. For correct code infilling, the input is the prefix and suffix after removing the buggy code hunk. In order to fill in the correct code, both the prefix and suffix can provide useful information. As a result, generative models are not suitable for



```
                …
primes = []
for n in range(2, max + 1):
            <INFILL>
        primes.append(n)
return primes
                …
```

```
                …
primes = []
for n in range(2, max + 1):
```

Fig. 3: APR input for single line generation

this task since the generation process conditions only on the context to the left (prefix). Therefore, for correct code infilling, we only use infilling models which perform generation by conditioning on both left (prefix) and right (suffix) code.

Figure 2 shows an example input for the infilling task. We start with the target buggy function we want to fix and remove the buggy code hunk. This gives us the prefix and suffix code which are still correct. We then place an infilling token between the prefix and suffix. This infilling token (e.g., `<INFILL>`) indicates to the model that this is the location where we want the new code to be generated at. The model then generates only the code to fill in the missing chunk and we obtain a patch by combining the model output with the prefix and suffix code snippets.

3) Single line generation: In single line generation, the buggy location is provided and the bug requires only a single line change. Figure 3a shows a similar setup to correct code infilling where we provide both the prefix and suffix code and use infilling models to generate a replacement line. Different from correct code infilling, we can also use generative models by providing only the prefix. Figure 3b demonstrates the setup to use generative models for this task. Since we know the bug requires only a single line change, we can stop the generation after the model has provided us with one line. We cannot apply the same strategy using generative models for correct code infilling since those bugs may need multiple lines to fix and we do not know when we can stop the generation [31]. Additionally, when using generative models for single line generation, we cannot provide the models with the suffix code due to the causal nature of the generative models. We contrast this with infilling models on the same task to demonstrate the effect of including the suffix context for APR.

C. Patch Ranking and Validation

For all 3 repair tasks, the patch generation process is similar – we provide the LLMs with the constructed input and use sampling to generate multiple patches per bug. We use nucleus sampling [57] with a sampling temperature. A lower temperature means the model is likely to pick tokens with higher likelihood, resulting in samples that are more similar (temperature of 0 gives deterministic result by picking the most likely token at each generation step). A higher temperature gives more probability for the model to pick a token with a lower likelihood, leading to more unique and interesting samples. How to pick an optimal temperature value is not obvious for a problem such as APR. For certain bugs, one may prefer a lower temperature value in order to quickly arrive at a reasonable patch. For harder bugs, a higher temperature value can be useful to generate more unique patches in an attempt to provide a fix. For our experiments, we use the default setting used in previous work [28], [33].

In addition to generating patches, we also record the entropy value of each patch. Entropy captures how *natural* [47] the generated sample is according to the model and can be calculated as the negative log probability of each generated token. Let $t_1, t_2, ..., t_n$ be the list of tokens generated and p_{t_i} be the model probability of generating token t_i given the previous context and generated tokens. Entropy is defined as:

$$mean\_entropy = -\sum_{i=1}^{n} \frac{\log(p_{t_i})}{n} \tag{1}$$

$$sum\_entropy = -\sum_{i=1}^{n} \log(p_{t_i}) \tag{2}$$

Mean entropy averages entropies of all tokens generated whereas sum entropy computes the total entropy of the sequence. For patch ranking, we prioritize patches with lower entropy first. In this way, patches that are more *natural* [47] can be ranked higher. Previous work on leveraging LLMs for APR either used mean entropy [26] or sum entropy [31] without thorough evaluation, and mainly focused on patch ranking. In contrast, in this work, we empirically compare both entropy computations, and have further applied them for patch correctness checking [58]. Finally, for each patch generated, we filter out any patches with syntactic or semantic errors and validate the rest against the test suites to identify patches which pass all the tests.

IV. Experimental Setup

A. Research Questions

We study the following research questions:

- **RQ1: How do different types of LLMs perform for different APR settings?** We study the effectiveness of different LLMs on different repair datasets, across different languages and on different APR tasks. Furthermore, we evaluate the scaling behavior of LLMs when increasing model size with respect to APR ability, computation time and compilation rates to holistically evaluate each LLM.
- **RQ2: How does directly applying LLMs for APR compare against state-of-the-art APR tools?** We compare the results using LLMs against state-of-the-art baselines. We study the unique bugs fixed by LLMs and highlight the advantages of directly applying LLMs for APR.
- **RQ3: Can LLMs be directly used for patch ranking and correctness checking?** We use the built-in naturalness metric of LLMs (entropy) to evaluate if LLMs considers patched functions to be *more natural* than buggy functions and if entropy can directly rank the patches for patch ranking and correctness checking.
- **RQ4: Can we further improve the performance of LLMs?** We explore two directions for further improving LLMs' performance for APR: 1) increasing the number of samples, and 2) combining LLMs with templates.

B. Implementation

We implement the generation pipeline in Python using PyTorch [59] versions of each LLM. We use the Hugging Face [48] to load the model weights and generate outputs. For Codex, we use API access provided by OpenAI to query the model [60]. To use Codex for correct code infilling, we append the API request with an additional suffix parameter [55] with

TABLE II: Evaluation dataset statistics

Dataset	#Bugs	#SF	#SH	#SL	Source	Language
Defects4J 1.2	391	255	154	80	real-world	Java
Defects4J 2.0	438	228	159	78	real-world	Java
QuixBugs-Java	40	40	37	36	coding problems	Java
QuixBugs-Python	40	40	40	40	coding problems	Python
ManyBugs	185	39	23	12	real-world	C
Total	1094	572	413	246		

the extracted suffix from the bug. For all our experiments, we directly reuse the weights of each model. Our default setting for generation uses nucleus sampling [57] with top p = 0.95, temperature = 0.8 and 200 samples per bug. This generation setting is consistent with previous studies on LLMs [28], [31], [33]. Patches are generated on a 32-Core workstation with Ryzen Threadripper PRO 3975WX CPU, 256 GB RAM and NVIDIA RTX A6000 GPU, running Ubuntu 20.04.4 LTS.

C. Subject Systems

For evaluation, we use 5 APR benchmarks spanning across 3 programming languages. We focus on bugs where the fix is within a single function, which is also the focus of most recent APR work [21], [22], [24], [61]. To this end, we filter these benchmarks to find bugs that fit our designed repair settings. Table II presents the details of each repair dataset. Column **Dataset** is the dataset name, **#Bugs** is the total number of bugs, **#SF, #SH, #SL** shows the number of bugs which the reference fix is within a single function, single hunk (consecutive lines) and single line. **Source** refers to where the bugs are collected from, **Language** is the programming language of the bugs. We next discuss the detailed dataset information:

1) Defects4J 1.2 and 2.0 [62]: The most widely studied APR benchmark with a collection of bugs gathered from open-source projects in Java containing pairs of buggy and patch versions of the source project. Since Defects4J has been updated to include more bugs from additional projects, we consider 2 different versions of Defects4J. Defects4J 1.2 contains 391 bugs (removing the 4 depreciated bugs) from 6 open-source Java projects. Defects4J 2.0 contains 438 new bugs from 9 additional projects. Each bug in Defects4J also contains developer tests exposing the bug.

2) QuixBugs-Python and -Java [63]: A multi-lingual repair benchmark with 40 classic programming problems. QuixBugs benchmark is constructed from a programming challenge where programmers were asked to fix a small buggy function. QuixBugs was originally in Python but has been translated to Java, with both versions having the same 40 bugs. Each bug is accompanied with multiple test inputs and expected outputs.

3) ManyBugs [64]: A C repair dataset consisting of 185 bugs gathered from 9 open-source projects with developer written tests. Each bug is manually verified and classified into a bug type. However, we were not able to reproduce all bugs from the dataset (i.e., builds successfully and reference patches can pass all provided tests). As such we only use the 91 bugs where the results were reproducible by us.

TABLE III: Complete function APR (SF bugs)

Dataset	GPT-Neo 125M	GPT-Neo 1.3B	GPT-Neo 2.7B	GPT-J	GPT-NeoX	Codex
Defects4J 1.2	6 / 8	7 / 16	10 / 24	14 / 31	18 / 36	63 / 102
Defects4J 2.0	2 / 17	4 / 18	6 / 20	11 / 33	15 / 36	49 / 93
QuixBugs-Java	1 / 3	4 / 5	3 / 5	3 / 5	8 / 9	32 / 35
QuixBugs-Py	1 / 3	4 / 6	4 / 6	13 / 17	19 / 22	37 / 37
ManyBugs	0 / 2	1 / 4	2 / 4	3 / 6	4 / 12	7 / 15

TABLE IV: Correct code infilling APR (SH bugs)

Dataset	CodeT5	InCoder 1.3B	InCoder 6.7B	Codex
Defects4J 1.2	6 / 13	32 / 51	37 / 53	62 / 77
Defects4J 2.0	12 / 19	31 / 56	37 / 61	56 / 85
QuixBugs-Java	10 / 10	21 / 26	26 / 29	34 / 36
QuixBugs-Py	7 / 8	25 / 26	27 / 28	39 / 39
ManyBugs	2 / 5	8 / 12	9 / 13	12 / 15

D. Compared Techniques

We compare against the state-of-the-art APR baselines with both learning-based and traditional APR tools. We choose 8 recent learning-based APR tools: AlphaRepair [26], RewardRepair [23], Recoder [21], DeepDebug [65], CURE [22], CoCoNuT [24], DLFix [66] and SequenceR [67]. Apart from AlphaRepair, these learning-based APR baselines are based on the NMT models. AlphaRepair combines a LLM (CodeBERT) with simple templates to generate patches under a zero-shot setting. Furthermore, we also choose 12 traditional APR tools: TBar [19], PraPR [20], AVATAR [16], SimFix [68], FixMiner [15], CapGen [9], JAID [69], SketchFix [13], NOPOL [12], jGenProg [70], jMutRepair [14], and jKali [14]. In total, we evaluate against 20 different APR tools. We compare against the baseline results on Defects4J 1.2, 2.0, QuixBugs-Python and Java on perfect fault localization - the ground-truth fix location is known to the repair tool. This is the preferred comparison setting as it eliminates the impact of differences in fault localization have on the result [21], [22], [24], [71]. Due to the lack of recent APR tools that are evaluated on ManyBugs, we only use it for RQ1. We follow prior work [19]–[22] and directly use the correct patch results from previous studies [19], [20], [26].

E. Evaluation Metrics

To evaluate the repair performance, we use the standard metrics of *plausible patches* – passing the all test cases, and *correct patches* – syntactically or semantically equivalent to the reference patches. To determine correct patches, we follow the standard practice in APR research and manually inspect each plausible patch for semantic equivalency.

V. Result

A. RQ1: Comparison of Different LLMs

1) Repair effectiveness: We first compare LLMs against each other in generating plausible and correct patches. Table III shows the results of 6 generative models under complete function generation setting. The two integers in each cell represent the number of correct and plausible patches. We first observe that similar to previous studies in NLP [40], there is a scaling effect on the repair effectiveness. *As we increase the size of the model, we also increase in the number*

TABLE V: Single line APR (SL bugs)

Dataset	GPT-Neo 125M	GPT-Neo 1.3B	GPT-Neo 2.7B	GPT-J	GPT-NeoX	CodeT5	InCoder 1.3B	InCoder 6.7B	Codex single-line	Codex suffix
Defects4J 1.2	5 / 10	12 / 20	13 / 21	16 / 26	21 / 31	5 / 12	21 / 36	26 / 38	32 / 37	39 / 47
Defects4J 2.0	8 / 17	10 / 26	16 / 28	12 / 26	19 / 36	9 / 15	15 / 32	21 / 37	26 / 38	31 / 45
QuixBugs-Java	8 / 9	19 / 20	16 / 17	20 / 21	20 / 21	10 / 10	21 / 26	26 / 29	30 / 31	34 / 36
QuixBugs-Python	9 / 10	14 / 14	22 / 23	26 / 27	28 / 28	7 / 8	25 / 26	27 / 28	36 / 36	39 / 39
ManyBugs	2 / 4	2 / 5	3 / 5	6 / 7	6 / 9	2 / 4	8 / 11	9 / 11	8 / 10	10 / 11

of correct and plausible patches generated. Directly looking at the group of GPT models trained on the same dataset, we see that the performance consistently increases as we use larger models across all repair datasets. However, we see that the Codex model (12B) outperforms the biggest model (GPT-NeoX (20B)). We hypothesize that this is because Codex is designed and finetuned for code generation; on the other hand, while the training dataset of GPT-NeoX is partially made up of code, it is designed for general purpose text generation.

Tables IV and V show the results on the correct code infilling and single line generation repair tasks. Similar to the previous result, we again see the scaling effect of increased performance as model size increases. Compared to complete function generation, we observe that each model using correct code infilling and single line generation is able to produce a higher ratio of correct fixes to the total number of bugs. Furthermore, we also observe that the ratio of correct patches to plausible patches is higher in the latter 2 settings as well. This signals that patches produced using code infilling and single line generation is more likely to be the correct fix. The improved performance is because for complete function generation the model needs to understand the prompt given (Section III-B1), localize the bug and provide the correct fix. On the other hand, when we provide the model with the buggy location information in correct code infilling and single line generation, it only needs to fill in or complete the partial code, leading to more correct patches. This comparison is more direct when evaluating the Codex model, the only model that can perform both code infilling and function generation. We see that when performing correct code infilling, Codex is able to fix 40% (62/154) of the total bugs whereas when asked to generate the entire function, it drops to 28% (63/225).

For single line generation results in Table V, we included both generative and infilling models. However, for generative models we are not able to provide it with suffix code snippets since their generation is dependent only on the previous context. We compare this with infilling models, which can perform infilling conditioned on both the context before and after. We observe that infilling models perform better than their generative counterparts. Additionally, since we are able to use both the generative and infilling versions of Codex, we can directly compare the repair ability of the model when given only the prefix versus both prefix and suffix context. We see that when using the suffix information from the original buggy function, the Codex model is able to improve the number of correct and plausible fixes across all repair datasets. This shows that for repair, *successfully utilizing the code after the buggy lines is important for fixing bugs.*

TABLE VI: Patch generation speed (#patch/min)

Models	Defects4J 1.2			QuixBugs-Python		
	CF	CI	SL	CF	CI	SL
GPT-Neo 125M	139	-	1080	369	-	1061
GPT-Neo 1.3B	31	-	543	127	-	814
GPT-Neo 2.7B	27	-	489	85	-	625
GPT-J	15	-	227	39	-	354
GPT-NeoX	2	-	47	6	-	73
CodeT5	-	969	-	-	1991	-
InCoder 1.3B	-	535	-	-	1083	-
InCoder 6.7B	-	288	-	-	419	-

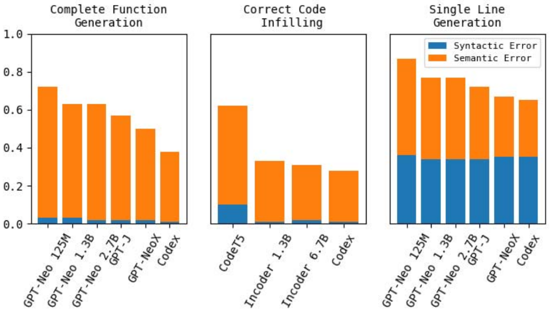


Fig. 4: Syntactic and semantic error rates on Defects4J 1.2

2) Speed: Next we look at the speed of patch generation using LLMs. We already saw from the previous result analysis that as we increase the size of the model, we obtain an increase in repair performance. However, such performance increase does not come for free as larger models require longer time for inferencing. Table VI shows the samples generated per minute for different LLMs on Defects4J 1.2 and QuixBugs-Python with the 3 repair generation settings (Columns CF, CI, SL refer to complete function, correct infilling and single line generation, respectively). We only include models that we run locally on the same hardware (i.e., excluding Codex since it is only accessible through API access). We first observe that as we increase model size, the patch generation speed drastically slows down (71x slower on GPT-NeoX than GPT-Neo 125M on complete function generation). This demonstrates the trade-off between repair effectiveness and time cost when using large models. Additionally, we see that compared to single line generation and correct code infilling, complete function generation takes significantly more time, since generating an entire function is much more time consuming than generating a single line or hunk. This shows *while LLMs have the capability to perform fault localization and repair in one shot, for real-world software systems, it is still more cost-effective to first use traditional fault localization techniques [46] to pinpoint the precise bug locations and then leverage LLMs for more targeted patch generation.*

TABLE VII: Defects4J 1.2 baseline comparison

Tools / Models	Single func. (255 bugs)	Patch func.	Correct hunk	Single line
AlphaRepair	67			
RewardRepair	48			
Recoder	61			
TBar	54			
CURE	52			
GPT-Neo 125M	9	6	-	5
GPT-Neo 1.3B	18	7	-	12
GPT-Neo 2.7B	20	10	-	13
GPT-J	28	14	-	16
GPT-NeoX	34	18	-	21
CodeT5	6	-	6	-
INCODER 1.3B	32	-	32	-
INCODER 6.7B	37	-	37	-
Codex	99	63	62	32
Total	109	69	74	40

3) Compilation rate: We evaluate the compilation rate of the patches generated by each LLM. Figure 4 shows the syntactic and semantic error rates of all studied LLMs using the three repair settings on Defects4J 1.2. We first observe that the overall error rate (syntactic + semantic) of the generated patches goes down as we increase the size of the model. This reaffirms the previously discussed scaling effect of LLMs and show that the patches generated by larger models contain less errors. Next we see that all generative models using single line generation produced a high number of syntactic errors. Recall that single line generation when using generative models only provides the prefix in the buggy function. As a result, the generated line can easily introduce some syntax errors (e.g., adding an if statement with an opening bracket) since the model does not know what the suffix code context is. On the other hand, the amount of syntax errors produced in the two other settings are much lower. For complete function generation, LLMs can effectively retain the syntax of the language during training and generate syntactically correct functions. *For correct code infilling, not only do we get low syntactic errors but also achieve the lowest semantic errors.* Having both the prefix and suffix provides the model with sufficient context which leads to higher compilable patch rate.

B. RQ2: Comparison against State-of-the-art APR tools

1) Defects4J 1.2 results: We first compare the results of directly using LLMs for repair against both traditional and learning-based APR tools on Defects4J 1.2. Table VII shows the number of correct bug fixes of the top baseline tools and also the LLMs in our evaluation. The last 3 columns present the number of correct patches generated when using each of the three APR settings. We then combine all patches generated for each of the models together (Column 2) to demonstrate the total number of fixes that can be obtained for the 255 single function bugs in Defects4J 1.2. Note that this is still a fair comparison – prior APR techniques typically use a timeout of 5h for each bug [21], [22], [26], while generating 200 patches for each of the 3 settings (i.e., at most 600 patches in total) costs no more than 2.5 hours for each model.

We observe that some of the models are able to achieve comparable performances compared to some of the recent state-of-the-art APR tools. Additionally, this result is obtained while generating only up to 600 samples per bug whereas prior approaches, especially learning-based tools, can generate up to 5000 patches per bug [22], [24], [26]. While the most effective model (Codex) can already outperform all existing techniques (e.g., fixing 99 single-function bugs), by combining the patches generated by all models (Total), we can achieve 109 correct fixes on single function bugs! The surprising results show that *by directly applying LLMs for APR without any specific change/finetuning, we can already achieve the highest number of correct fixes compared to existing baselines.*

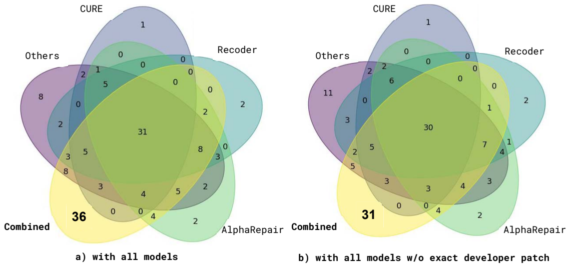
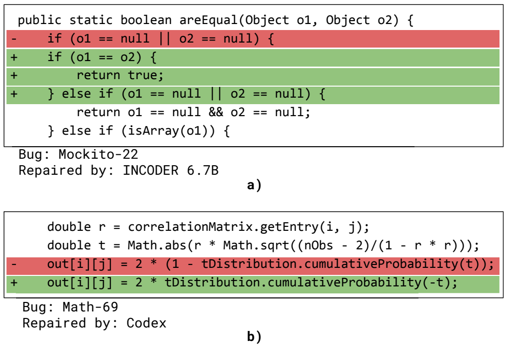


Fig. 5: Bug fix Venn diagram on Defects4J 1.2



```
public static boolean areEqual(Object o1, Object o2) {
-    if (o1 == null || o2 == null) {
+    if (o1 == o2) {
+        return true;
+    } else if (o1 == null || o2 == null) {
         return o1 == null && o2 == null;
     } else if (isArray(o1)) {
```

Bug: Mockito-22
Repaired by: INCODER 6.7B

a)

```
     double r = correlationMatrix.getEntry(i, j);
     double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
-    out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(t));
+    out[i][j] = 2 * tDistribution.cumulativeProbability(-t);
```

Bug: Math-69
Repaired by: Codex

b)

Fig. 6: Unique bug fixes generated by LLMs

Figure 5 presents the Venn diagram of unique fixes that can be generated using LLMs compared to the 3 best performing baselines on all the single function bugs in Defects4J 1.2. We also combine all fixes from other baselines together into the "Others" category in the Venn diagram. We observe that by combining all the models together, we can generate a significant amount of unique bug fixes (36) that no other tools have fixed so far. Due to the potential data leakage issue (discussed in detail in Section VI), we further investigate whether LLMs can generate correct patches that are not exactly the same as developer patches. Figure 5b shows the unique bug fixes on Defects4J 1.2 compared to the baselines when we remove all fixes which are exactly the same as the developer patch. We observe that combining all LLMs together would still achieve the highest number of bug fixes (93) with 31 unique bug fixes.

To demonstrate the ability of these LLMs, we show some unique fixes produced by them. Figure 6a is a correct patch produced by the INCODER 6.7B model under correct code infilling task. We see here that the function is called `areEqual` and the bug is caused by missing a specific case of comparing

TABLE VIII: Additional baseline comparison

Tools / Models	Defects4J 2.0 (78 bugs)	QuixBugs-Java (40 bugs)	QuixBugs-Python (40 bugs)
AlphaRepair	35	28	27
RewardRepair	25	20	-
DeepDebug	-	-	21
Recoder	11	17	-
CURE	-	21	-
TBar	8	-	-
CoCoNuT	-	13	19
GPT-Neo 125M	10	8	9
GPT-Neo 1.3B	11	20	17
GPT-Neo 2.7B	19	18	24
GPT-J	16	22	29
GPT-NeoX	24	21	31
CodeT5	9	10	7
InCoder 1.3B	15	21	25
InCoder 6.7B	21	26	27
Codex	45	38	40
Total	52	38	40

TABLE IX: Mean entropy of generated patches

	Models	Defects4J 1.2			QuixBugs-Python		
		C	P	NP	C	P	NP
Function Gen.	GPT-Neo 125M	0.08	0.13	0.23	0.10	0.10	0.20
	GPT-Neo 1.3B	0.12	0.12	0.19	0.06	0.05	0.09
	GPT-Neo 2.7B	0.09	0.13	0.17	0.05	0.06	0.08
	GPT-J	0.07	0.10	0.12	0.04	0.05	0.08
	GPT-NeoX	0.08	0.11	0.13	0.05	0.07	0.10
	Codex	0.04	0.05	0.08	0.11	0.13	0.16
Infilling	CodeT5	0.50	0.51	0.54	0.51	0.50	0.59
	InCoder 1.3B	0.49	0.58	0.65	0.54	0.56	0.65
	InCoder 6.7B	0.45	0.50	0.61	0.61	0.60	0.65
	Codex	0.43	0.43	0.50	0.32	0.33	0.42
Line Gen.	GPT-Neo 125M	0.38	0.42	0.58	0.41	0.45	0.61
	GPT-Neo 1.3B	0.32	0.38	0.58	0.25	0.27	0.47
	GPT-Neo 2.7B	0.28	0.32	0.55	0.21	0.26	0.40
	GPT-J	0.29	0.33	0.54	0.20	0.22	0.38
	GPT-NeoX	0.39	0.42	0.71	0.26	0.28	0.55
	Codex	0.19	0.28	0.57	0.18	0.23	0.60

if the two inputs have the same reference. Using both the prefix (name of the function) and suffix (other comparison statements with return values), the model figures out the correct code to be inserted here (first checking if the references are the same before proceeding). Such code is commonly found in open-source projects which use similar comparison functions where the LLMs can learn from. In fact, we found several similar comparison functions (checking if the objects have the same reference) [72]–[75] in different projects as a part of The Pile dataset [51] that some of the LLMs were trained on. Furthermore, unlike traditional APR tools which often work on a single line, LLMs can generate multiple lines of code in order to provide the correct fixes.

Figure 6b shows a patch of the Math-69 bug generated by Codex. The function here calculates a matrix of p-values of a 2-sided, 2-sample t-test. The bug is caused by precision error when the function call is extremely close to 1. Here the model generates an alternative way of calculating the p-value which is much more stable than before. This is a hard bug to fix since the change is quite subtle but it does not fit any of the common templates used in traditional APR. To generate the correct fix, the model needs to understand the goal of the function (p-value calculation) and use statistical formulas. Both of which can be achieved by Codex as it is trained not only on code but also on general text, which contains many descriptions and examples of t-test p-value calculations. This unique fix shows the benefit of using LLMs for program repair where domain knowledge of the project can be utilized as well.

2) Additional results: In addition to comparing against state-of-the-art baselines on Defects4J 1.2, we also compare the performance of LLMs on other datasets widely used to evaluate previous APR tools. Table VIII shows the results on Defects4J 2.0, QuixBugs-Java and -Python where we also combine the correct bug fixes of the 3 generation strategies together. Similar to the Defects4J 1.2 results, we observe that many models can achieve similar (or even better) performance with carefully designed APR tools. More surprisingly, *all 9 studied LLMs can outperform TBar, state-of-the-art template-based APR tool, and are competitive compared with the recent Recoder technique on the Defects4J 2.0 dataset.* Furthermore, unlike many baselines which can only be used on a single language (specifically designed for a particular language or requiring additional finetuning on another language), the LLMs can be directly applied for multi-lingual repair.

C. RQ3: Patch Ranking and Correctness Checking Analysis

1) Entropy: As we are using LLMs for patch generation, this allows us to compute the entropy of each patch. Entropy calculates how natural the generated sample is (Equation 1). Table IX shows the mean entropy values for correct (**C**), plausible (**P**) and non-plausible patches (**NP**). Each row shows the results of a LLM on a repair scenario containing bugs for which the LLM can produce a correct patch. We observe that average entropy value of correct and plausible patches for all models are less than non-plausible patches. Although not shown in the table, we observe the same finding when comparing patches using sum entropy. In other words, *the studied LLMs consider correct patches which correctly fix the underlying bugs to be more nature than other patches.* Additionally, while the entropy difference between correct and plausible patches is not as drastic as compared to non-plausible patches, we also find that correct patches are in general less entropic than plausible ones. Recent work [58] has shown that existing solutions for *patch-correctness checking* (i.e., identifying correct patches from plausible patches) can suffer from dataset overfitting and performance drops when applied on more complicated patches. We demonstrate for the first time that *entropy computation via LLMs can help distinguish correct patches from plausible patches*, indicating a promising future of directly leveraging the LLM entropy metric for patch-correctness checking.

2) Patch ranking: Using the entropy values of each generated patch, we perform ranking to validate patches with higher rank (lower entropy) first. We pick 5 LLMs with the highest number of correct patches to perform this analysis. Figure 7 shows the number of bugs fixed for the Defects4J 1.2 dataset using different patch ranking strategies as we increase the number of patches to validate. We see that compared to randomly picking patches to validate (blue line), when using

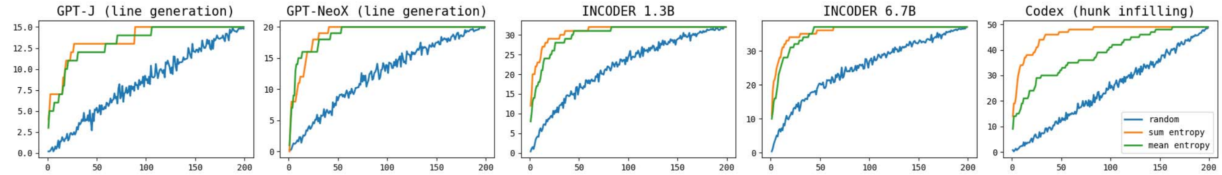


Fig. 7: Number of bugs fixes when using different patch ranking strategies on Defects4J 1.2

TABLE X: Further improving LLM-based APR

Tools / Models	Defects4J 1.2 All	Defects4J 2.0 Single Line	QuixBugs-Python
AlphaRepair	74	35	27
RewardRepair	50	25	-
DeepDebug	-	-	21
Recoder	65	11	-
TBar	68	8	-
InCoder (200)	37	21	27
InCoder (2000)	64	25	32
InCoder w/ template (2000)	**78**	**39**	**37**

entropy rankings (orange and green line), we can validate the correct patches faster. This shows that entropy can be an effective measure used to rank the potential patches to prioritize lower entropy patches for validation under tighter time constraints. Furthermore, we observe that *sum entropy performs slightly better compared to mean entropy*. We hypothesize that this is because sum entropy calculates the entire sequence entropy regardless of the length of the generated sequence. As such, shorter sequences tend to have lower sum entropy compared to longer sequences; interestingly, this is consistent with traditional APR or patch correctness checking techniques [11], [76], [77], which favor simple patches over complicated ones following the Occam's razor hypothesis [78].

D. RQ4: Improvements on Direct LLM-based APR

In previous RQs, we showed that by directly applying LLMs for APR we can already achieve comparable performance with previous APR tools. We further explore the possibilities to boost the ability of LLMs for APR. For this experimental setup, we choose the best performing model (apart from Codex, which already outperforms existing APR techniques without any further extension) – InCoder 6.7B and run the model longer (2000 samples per bug) combined with repair templates. We evaluate on all bugs in Defects4J 1.2 by adjusting our infilling-style repair settings, following AlphaRepair [26] (which demonstrated the best performance among all settings in our study), to generate patches for every location which is changed by the reference patch instead of only on a single change location. This setup is similar to previous learning-based repair tools [21], [26] and allows us to compare on the full Defects4J 1.2 dataset. Furthermore, following prior work [26], we include evaluation on Defects4J 2.0 single line bugs and QuixBugs-Python.

Table X shows the baseline tools along with our model setups. **InCoder (200)** is our default setup from previous evaluation that generates 200 samples per bug. **InCoder (2000)** shows the results when we increase the number of samples to 2000. **InCoder w/ template (2000)** contains the results when combining repair template with the InCoder model. Following the AlphaRepair baseline, we apply different repair templates by using the original buggy line. Such templates include: keeping parts of the prefix or suffix, replacing method calls or parameters, and changing/adding new boolean conditions or operators to the buggy line. These repair templates make use of the original buggy line and provide important starting code for the model.

We observe that if we *apply the model longer and generate more samples, we can drastically improve the number of correct bugs fixed* in all three datasets and achieve very close result to that obtained by the best baseline. Moreover, we can obtain further improvements by using simple repair templates and achieve the highest number of correctly fixed bugs on all datasets, e.g., fixing 78 bugs on Defects4J 1.2 with 15 unique bug fixes that no other baseline tools have fixed before. This finding shows that not only can LLMs be effective when directly used for program repair, *we can combine them with more domain specific techniques such as simple repair templates to further improve their performance.*

VI. Threats to Validity

Internal. One internal threat to validity comes from our manual validation of plausible patches to determine semantically correct patches. To address this, we carefully performed the analysis and released the correct patches and code used to perform the experiments for public evaluation [79].

Another internal threat comes from the potential data leakage of real developer patched functions being part of the original training data. To address this, we examine the patches LLMs generated for Defects4J 1.2 since this is the most widely studied dataset for APR and we mainly compared with state-of-the-art APR tools on this dataset. We first check if the bugs fixed by each LLM contain correct patches different than the reference developer patches. Out of the 354 individual bug fixes by all models on Defects4J 1.2, 234 fixes (66%) contain a patch that is different than the developer patch. We also found that due to the simplicity of single line patches, majority of the correct patches generated for single line bugs are the same as the developer patch. If we exclude single line bugs, the percentage increases to 77% (196/255). Out of the 109 bugs that can be fixed by combining all correct patches generated by all LLMs together (Total row in Table VII), 93 bugs (85%) are fixed by at least one correct patch that is different than the original developer patch, e.g., as shown in Figure 5b, removing LLM fixes that are exactly the same as the developer patches can still fix 31 bugs that prior tools cannot fix.

Since we only have access to the training data used in CodeT5, GPT-Neo, GPT-J and GPT-NeoX models, we further check if the fixed function is within the training datasets when the correct patch is equivalent to the developer fix for these models. We found that while 38% (48/128) of bugs fixes contain only the same fix as the developer patch, only 15% (20/128) of those patches are also found in the original training data, showing that the majority of correct bug fixes provided by these LLMs are not simply from memorizing the training data. Moreover, our RQ4 shows that improvements can be further made by combining repair templates with LLMs, which is orthogonal to the data leakage issue. Additionally, We observe that LLMs are able to achieve the state-of-the-art results on QuixBugs dataset which is not part of the training data as it has low number of stars on GitHub and contains synthetic bugs and patches that are not part of any larger real-world projects. Further reducing the data leakage issue would require retraining the LLMs, which could be extremely costly.

External. We evaluate LLMs on 5 repair datasets across 3 programming languages, making our evaluation one of the most comprehensive studies in APR. However, our findings may still not generalize to other datasets or languages.

VII. Discussion and Future Work

In this work, we conduct a large-scale study on directly applying LLMs for APR, one of the most important problems in Software Engineering (SE). We demonstrate not only by directly applying LLMs we can already outperform prior APR techniques studied for over a decade, but also that we can further boost LLM performance by combining domain-specific techniques from SE. Building on these findings, we highlight two key directions for future work:

Improving LLM performance for APR. We plan to continue improving the performance of LLMs for APR. One approach is to use the additional information from project-specific knowledge (i.e., from buggy project itself following the plastic surgery hypothesis [80]). For example, one could fine-tune the LLMs slightly on the original buggy project to *prime* the model to generate code that fits the style/convention used in the project. Another approach is to incorporate repair-specific knowledge by using additional templates as demonstrated in Section V-D to reduce the amount of code LLM has to generate and arrive at the correct patch faster. Along with these potential improvement directions, we also believe that we can use other new types of LLMs (i.e., dialogue-based) for APR such as ChatGPT [29]. ChatGPT is fine-tuned using reinforcement learning algorithm with human feedback designed for dialogues/conversations. We can leverage the currently underused testcase result to provide feedback to ChatGPT in a conversational manner and allow the model to correct its previous mistakes and generate more correct patches [81].

Application of LLMs for other relevant SE tasks. While we study the performance of LLMs for APR, LLMs can be used for various other software engineering tasks. One such task is fuzzing [82], where LLMs can be potentially used to help generate arbitrary inputs to fuzz test various software systems. Compared with traditional automated fuzzing techniques [83] which require extensive human efforts for ensuring the syntactic/semantic validity of input generation/mutation, LLMs offer an alternative solution by learning from billions of available code snippets in the wild to generate syntactically and also semantically valid input programs fully automatically [84]. LLMs can also be used to target more context dependent tasks such as test [85] or test-oracle [86] generation where the input can be the focal method we want to generate unit test for. While existing learning-based test-oracle generation techniques [87], [88] mainly formulate the problem as a classification or NMT problem, a more natural solution is to leverage the LLMs to directly complete or infill the oracles based on context information (such as focal method and test prefix/suffix). Similar to APR, mutation testing [89] also applies systematic modifications to programs under test. As a result, it is very natural to directly apply infilling-style APR techniques (such as AlphaRepair [26]) for mutation testing. In addition to these discussed SE tasks above, we believe our study results and techniques can also motivate, inspire, and be applied to many other relevant SE tasks involving code generation/mutation. These potential applications along with LLMs for APR highlight the promising future of using LLMs to help with SE in general.

VIII. Conclusion

We present an extensive evaluation on LLMs for automated program repair. We use 9 state-of-the-art LLMs with 5 different repair datasets and design different practical repair settings to compare and contrast the repair effectiveness of different LLMs. In our evaluation, we shed light on the scaling effect that increasing model size has on various important factors in APR such as the number of bugs fixed, the speed of patch generation, and the compilation rate. Also, we compare the performance of LLMs against state-of-the-art APR tools and highlight the unique fixes and advantages of using LLMs for APR. Furthermore, we evaluated the ability for LLMs to perform patch ranking and patch correctness checking in order to prioritize correct patches for faster repair. Lastly, we demonstrate the possibilities (i.e., increasing the sample size and combining LLMs with repair templates) to further boost the performance of LLMs for APR. The results from our study demonstrate promising future of adopting LLMs for APR and beyond (e.g., other SE tasks involving program generation/mutation).

Acknowledgments

We thank the reviewers for their insightful feedback and comments to improve this paper. We thank Yifeng Ding for his helpful discussion on this work. This work was partially supported by NSF grants CCF-2131943, and CCF-2141474. We also acknowledge support from Kwai Inc. and Ant Group.

References

[1] K. Luzniak, "Software for the healthcare industry: what is it and why it's worth using?" *neoteric*, 2022, https://neoteric.eu/blog/software-for-the-healthcare-industry-what-is-it-and-why-its-worth-using.

[2] N. Mayersohn, "Data driving new approaches to transportation," *The New York Times*, 2022, https://www.nytimes.com/2020/02/05/technology/data-micromobility-electric-scooters-mds.html.

[3] E. Richards, "Software's dangerous aspect," *The Washington Post*, 1990, https://www.washingtonpost.com/archive/politics/1990/12/09/softwares-dangerous-aspect/9b2e9243-8deb-4ac7-9e8f-968de0806e5e/.

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

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

[6] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, 2019.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, 2012.

[8] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016.

[9] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016.

[11] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *ESEC/FSE*, 2017.

[12] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, 2014.

[13] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: A tool for automated program repair approach using lazy candidate generation," in *ESEC/FSE*, 2018.

[14] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *ISSTA*, 2016.

[15] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empir. Softw. Eng.*, vol. 25, 2020.

[16] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2019.

[17] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *ISSTA*, 2020.

[18] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: An extensive study on 16 program repair systems," in *ASE*, 2020.

[19] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *ISSTA*, 2019.

[20] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*, 2019.

[21] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/FSE*, 2021.

[22] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," *ICSE*, 2021.

[23] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *ICSE*, 2022.

[24] 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 *ISSTA*, 2020.

[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014, arXiv:1409.3215.

[26] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *ESEC/FSE*, 2022.

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

[28] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021, arXiv:2107.03374.

[29] J. Schulman, B. Zoph, J. H. Christina Kim, J. Menick, J. Weng, J. F. C. Uribe, L. Fedus, L. Metz, M. Pokorny, R. G. Lopes, S. Zhao, A. Vijayvergiya, E. Sigler, A. Perelman, C. Voss, M. Heaton, J. Parish, D. Cummings, R. Nayak, V. Balcom, D. Schnurr, T. Kaftan, C. Hallacy, N. Turley, N. Deutsch, V. Goel, J. Ward, A. Konstantinidis, W. Zaremba, L. Ouyang, L. Bogdonoff, J. Gross, D. Medina, S. Yoo, T. Lee, R. Lowe, D. Mossing, J. Huizinga, R. Jiang, C. Wainwright, D. Almeida, S. Lin, M. Zhang, K. Xiao, K. Slama, S. Bills, A. Gray, J. Leike, J. Pachocki, P. Tillet, S. Jain, G. Brockman, and N. Ryder, "Chatgpt: Optimizing language models for dialogue," 2022, https://openai.com/blog/chatgpt/.

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020, arXiv:2002.08155.

[31] S. D. Kolak, R. Martins, C. L. Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Deep Learning for Code Workshop*, 2022.

[32] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2022.

[33] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022, arXiv:2204.05999.

[34] 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," 2021, arXiv:2102.04664.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, 2002.

[36] Y. Liu, "Fine-tune bert for extractive summarization," 2019, arXiv:1903.10318.

[37] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," 2020, arXiv:1906.08237.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, arXiv:1706.03762.

[39] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," 2021, arXiv:2102.07350.

[40] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020, arXiv:2001.08361.

[41] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018, arXiv:1810.04805.

[42] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[43] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, 2020.

[44] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019, arXiv:1910.13461.

[45] A. Aghajanyan, B. Huang, C. Ross, V. Karpukhin, H. Xu, N. Goyal, D. Okhonko, M. Joshi, G. Ghosh, M. Lewis, and L. Zettlemoyer, "Cm3: A causal masked multimodal model of the internet," 2022, arXiv:2201.07520.

[46] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007.

[47] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE*, 2012.

[48] "Hugging face," 2022, https://huggingface.co.

[49] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[50] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, "GPT-NeoX-20B: An open-source autoregressive language model," in *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022, arXiv:2204.06745.

[51] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," 2020, arXiv:2101.00027.

[52] S. J. Yue Wang, Weishi Wang 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, EMNLP 2021*, 2021.

[53] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020, arXiv:1909.09436.

[54] "Bigquery github repos," 2022, https://console.cloud.google.com/marketplace/details/github/github-repos.

[55] "Codex suffix api," https://beta.openai.com/docs/api-reference/completions/create#completions/create-suffix, 2022.

[56] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," *ACM SIGPLAN Notices*, vol. 48, 2013.

[57] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," 2019, arXiv:1904.09751.

[58] Y. Wang, J. Yang, Y. Lou, M. Wen, and L. Zhang, "Attention: Not just another dataset for patch-correctness checking," 2022, arXiv:2207.06590.

[59] "Pytorch," 2022, http://pytorch.org.

[60] "Openai api," 2022, https://openai.com/api.

[61] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022, arXiv:2203.07814.

[62] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," ser. ISSTA, 2014.

[63] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," ser. SPLASH Companion 2017, 2017.

[64] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, 2015.

[65] D. Drain, C. B. Clement, G. Serrato, and N. Sundaresan, "Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons," 2021, arXiv:2105.09352.

[66] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *ICSE*, 2020.

[67] 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 Transaction on Software Engineering*, 2019.

[68] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018.

[69] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE*, 2017.

[70] M. Martinez, T. Durieux, J. Xuan, R. Sommerard, and M. Monperrus, "Automatic repair of real bugs: An experience report on the defects4j dataset," 2015, arXiv:1505.07002.

[71] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *ASE*, 2018.

[72] "jetbrick-template-2x object comparison code," 2022, https://github.com/subchen/jetbrick-template-2x/blob/def3107e2878aa5bee32ac2ba3be8e241fba4a64/src/main/java/jetbrick/template/parser/ast/ALU.java#L421-L448.

[73] "goclipse object comparison code," 2022, https://github.com/GoClipse/goclipse/blob/e135d3a69e6498e278521c2542cee3808bd1377d/plugin_tooling/src-util/melnorme/utilbox/core/CoreUtil.java#L28-L30.

[74] "teiid object comparison code," 2022, https://github.com/teiid/teiid/blob/21c93a6fd4be2528f95224f99905d74479862d1b/federate-common-core/src/main/java/com/metamatrix/core/util/EquivalenceUtil.java#L49-L57.

[75] "Groza object comparison code," 2022, https://github.com/IoT-Technology/Groza/blob/fbafceef53d646025046990ffbd89bf701c56b45/dao/src/main/java/com/sanshengshui/server/dao/util/mapping/JsonTypeDescriptor.java#L49-L58.

[76] M. Asad, K. K. Ganguly, and K. Sakib, "Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair," in *ICSME*, 2019.

[77] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *ASE*, 2017.

[78] E. Sober, *Ockham's razors*. Cambridge University Press, 2015.

[79] "Dataset," 2023, https://zenodo.org/record/7592886.

[80] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *ESEC/FSE*, 2014.

[81] C. S. Xia and L. Zhang, "Conversational automated program repair," 2023, arXiv:2301.13246.

[82] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," 2019.

[83] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, vol. 14, no. 3, p. 151–165, mar 1971.

[84] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Fuzzing deep-learning libraries via large language models," 2022, arXiv:2212.14834.

[85] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, 2012.

[86] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[87] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: a neural method for test oracle generation," in *ICSE*, 2022.

[88] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *ICSE*, 2020.

[89] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, 2010.