Evaluating the repair ability of LLM under different prompt settings

Xiaolong Tian
Southern University of Science and Technology
Shenzhen, China
12010712@mail.sustech.edu.cn

2024 IEEE International Conference on Software Services Engineering (SSE) | 979-8-3503-6851-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/SSE62657.2024.00053

***Abstract*—As the increase of complexity and scale of software, the number of software bugs increases as well, which causes huge financial losses and poses a great threat to our lives. To address this issue, researchers have devoted themselves to automated program repair(APR), which aims to release programmers from the time-consuming manual debugging. In the last few decades, lots of APR tools have been proposed , such as constraint-based , heuristic-based , template-based and learning-based APRs. Unfortunately, traditional APRs are confronted with numerous challenges. Neither do they deeply understand the program's semantics, nor they can fix many types of bugs in different scenarios. Recently, researchers directly applied Large Language Models (LLMs) for APR and demonstrated that directly applying state-of-the-art LLMs can already substantially outperform all existing APR techniques on many datasets. However, there is a lack of study that fully reveals which information is useful for LLM-based APR. There is neither a study that reveals how we can further increase the repair ability of large language models by better utilizing prompts. In this study, we conduct an extensive study to figure out how bug descriptions, fault locations and few-shot prompting affect the repair ability of LLM on Defects4J dataset. We found that 1) Concise bug description is critical for the model to fix bugs. 2) Fault locations can improve LLM's repair ability in most cases, but it does not always helpful. 3) Few-shot prompting is not beneficial for enhancing the understanding of LLM regarding the faulty program and we'd better not use few-shot prompting. We are fully convinced that our findings will provide not only some theory evidences but also some insights for future research.**

***Index Terms*—APR, LLM, Fault Location, Bug Report, Few-shot prompting**

I. Introduction

Software is a fundamental infrastructure to our lives today and has infiltrated every aspect of life. However, with the increase of complexity and scale of software, the number of software bugs increases as well, which causes huge financial losses [1], [2] and poses a great threat to our lives [3]. On top of that, manually debugging is a time-consuming and tedious process. Recent studies have shown that debugging activities usually account for about 50 percent of the overall development cost of software products [4], [5]. Thus, in the last few decades, a lot of researchers have dedicated themselves to Automatic program repair(APR), which have the potential to release developers from heavy debugging tasks.

Automatic program repair techniques automatically repair software systems by producing a fixed code given the original code and the corresponding buggy location. To solve this topic, lots of methods have been proposed, such as constraint-based [6]–[9], heuristic based [10], [11], template-based [12]–[14] and learning-based APRs [15]–[18]. Nevertheless, traditional APR tools face numerous challenges. Heuristic-based APR tools have been shown to work for large programs [19], but the technique cannot "synthesize" an appropriate expression from variables and constants [6]. Constraint-based APRs fully leverage existing optimization and solution algorithms. However, such methods are difficult to apply to complex, large-scale programs due to their reliance on symbolic execution and constraint solving techniques. Furthermore, the patches generated by constraint-based APRs are elliptical and unreadable. Template-based APR tools can generate relatively high-quality patches, but it can only fix limited types of bugs which are pre-defined by human experts. Learning-based APRs are designed to generate more types of bugs, but they are still restricted by their training data.

Recently, researchers directly applied Large Language Models (LLMs) for APR and demonstrated that directly applying state-of-the-art LLMs can already substantially outperform all existing APR techniques on many datasets [20]. This discovery is a milestone in the domain of APR because it not only breaks through many limitations of traditional tools , but also provides a feasible direction for future research on APR.

However, there is a lack of study that fully reveals to what extent can we enhance the repair ability of LLM by providing LLM with different information. We also desired to know which information is more important for LLM to repair the buggy program. Since LLMs for APR has a promising future, the study of how to better utilize LLMs is of great significance. Motivated by this, we conduct an extensive study to evaluate LLM's repair ability by providing LLM with different information. In detail, we divide the information to be provided into three categories : bug descriptions, fault locations and few-shot prompting(Using several examples in a prompt to show how to perform a task is often called few-shot learning). To sum up, this study makes the following contributions:

- We compare the repair performance of LLMs with and without bug descriptions in bug report. We conclude that the description of a bug is critical for LLM to fix a buggy program whereas too much irrelevant information may have a negative impact on model's repair ability. So we'd

979-8-3503-6851-2/24/$31.00 ©2024 IEEE
DOI 10.1109/SSE62657.2024.00053

better provide the model with concise descriptions.

- We compare the repair performance of LLMs with and without fault location. We found that fault locations can help LLM fix bugs by accurately pointing out where modifications are needed. However, directly labeling all fault locations doesn't always help LLM fix bugs. To figure out the reason behind this phenomenon, we manually analysis the feature of some buggy programs which can be fixed without fault locations but cannot be fixed with fault locations. Finally we list some reasons we found and give the corresponding examples.
- We evaluate the model's repair capabilities when we apply different k in k-shot learning to LLM. We conclude that few-shot learning does not make a contribution to the enhancement of model's repair capability and we suggest not use few-shot prompting strategy when we use LLM to fix bugs.

II. Related Work

A. Automated program repair

A buggy program is a program that behaves abnormally and we can't get the desired output with specific input. For example, Figure 1 shows an algorithm of Fibonacci sequence. This program is buggy because if we input 3, we get 4 as our input. However, the desired output is 5. The error occurrs in the recursive formula in line 3. The formula should be "return fib(n-1) + fib(n-2) " instead of "return fib(n-1) + fib(n-3) ". The correct program is shown in Figure 2.

Automated Program Repair (APR) aims to automatically identify, analyze, and fix bugs or defects in software code without human intervention. The inputs for an APR tool consist of a program with defects and a "specification" (such as a test suite or a formal specification) that outlines the intended functionality of the program. The output for an APR tool is a repaired program and the repaired program will be validated against the test suite. A successful repair would be a modification of the program such that it passes all the tests in the test suite. Over the last few decades, lots of APR techniques have been proposed. Among traditional APR techniques, Genprog [10] , a heuristic-based APR tool, is often regarded as the pioneer of APR and has been widely studied by many researchers. It first transforms the source program into an Abstract Syntax Tree (AST), and then iteratively applies crossover and mutation operators to randomly delete, add, and replace nodes in the AST. Latter, Nguyen pointed out that heuristic-based techniques cannot "synthesize" an appropriate expression from variables and constants and proposed a constraint-based APR tool [6]. Dongsun [13] pointed out that GenProg is possible to generate nonsensical patches since this technique basically relies on random program mutations.However,constraint based APR tool is challenging to apply to complex, large-scale programs due to its reliance on symbolic execution and constraint solving techniques.

```
int fib(int n){
  if(n > 2){
     return fib(n-1) + fib(n-3);
  }else{
     return 1;
  }
}
```

Fig. 1. an example of buggy function

```
int fib(int n){
  if(n > 2){
     return fib(n-1) + fib(n-2);
  }else{
     return 1;
  }
}
```

Fig. 2. the fixed version of figure 1

Template-based APR tools can generate higher quality patches, but it can only fix the bug types that are part of the templates. To generate more types of bugs, researchers proposed learning based APR tools, which are based on NMT [25] techniques, where the goal is to translate a buggy program into a fixed program. Nevertheless, these learning-based tools are still limited in terms of the type of fixes it can apply due to this reliance on the bug-fixing data.

As mentioned above, traditional APR tools face many limitations. The accuracy of the patches generated by traditional APR tools still fall short of the requirements for industrial applications. Furthermore, although traditional APR tools can fix some buggy programs, in practice, they often fail to provide a reasonable explanation for their repair results, indicating a lack of true comprehension of the program's semantics. Consequently, the patch generation process possesses a degree of randomness and lack of insight.

B. LLM for APR

Large language models (LLMs) are based on the Transformer architecture [22], pre-trained on massive text data, and demonstrate excellent performance in natural language processing(NLP) tasks, such as text summarization [23], text classification [24] and machine translation [25]. LLMs can be categorizes into 3 types: encoder-only, decoder-only, and encoder-decoder. Encoder-only LLMs (BERT [26]) use only the encoder component. Decoder-only LLMs (GPT [27]) contain only the decoder component to predict the next token output given all previous tokens. Encoder-Decoder(T5 [28]) models combine the usage of both encoder and decoder. Encoder-only and Encoder-Decoder models are trained using Masked Language Modeling (MLM) or Masked Span Prediction (MSP) objective, respectively, where a small portion of the tokens are replaced with either masked tokens or masked span tokens and the LLMs are trained to recover the masked out tokens based on bi-directional context. Decoder-only models are trained using Causal Language Modeling objective by

training to predict the probability of the next token given all previous left only context.

Recently, researchers [20] used 9 different recent LLMs on 5 different repair datasets across 3 different programming languages (Java, Python, and C) and demonstrated that directly applying LLMs can already outperform prior APR techniques studied for over a decade. Since LLMs possess strong semantic understanding capabilities, they can address many challenges that traditional APR tools face. What's more, researchers are endeavored to trying different approaches to enhance the repair capability of large models. AlphaRepair [29] proposes the first cloze-style (or infilling-style) APR approach, where the buggy code is removed and a LLM directly predicts correct code given the prefix and suffix context. It treats APR as a cloze or text infilling task instead of an NMT task. FitRepair [22] combines the direct usage of LLMs with two domain-specific fine-tuning strategies and one prompting strategy (via information retrieval and static analysis) for more powerful APR. ChatRepair [21] leverages previously incorrect and plausible patches and test failure information, which enables models to learn from past failures and generate higher quality patches. All the things indicate that LLM-based APR has a promising future and more researchers will devote themselves in this field. Nevertheless, which information is critical for LLM to fix bug is yet to be revealed. Therefore we conduct an extensive study to reveal which information is critical for LLM to fix bug and we are fully convinced that our experimental result will provide not only some theory evidences but also some insights for future research.

III. Approach

In this study, we explore what may improve LLM's repair ability and we leverage 3 possible useful information that might help LLM fix bugs : bug descriptions in bug report, fault locations and few-shot prompting . We divide the large language model into 32 groups and each group is under different setting, which will be explained in detail later. Our experiment has 3 stages: pre-processing, patch generation and patch validation.

A. Pre-processing

In the first stage, we extract some specific information from the Defects4j dataset. First, among 835 bugs in the Defects4J 2.0 dataset,we filtered out the buggy programs where errors occurred only within a single function. Then we extracted the buggy function of each buggy program by using a python script. Fixes may include multiple lines, but are scoped to a singular function. We choose function as a repair unit because function generally contains comprehensive semantic information and the input prompt for LLMs are restricted to only a few thousand tokens that may not suffice to capture the entire file or repository level information [32]. Finally we extract the bug report of each buggy program by calling command-line interfaces in a python script.

B. Patch generation

In this stage, we leveraged 3 information:

- Bug report. Bug report is a natural language description of program's erroneous behavior which contains information about what is wrong and where developers should fix the given bug, which is submitted to bug tracking systems (e.g. Bugzilla, JIRA) by users or developers and is beneficial for software maintenance [33]. It usually contains a bug id, an issue title that describes the bug at a high level, and an issue description that provides more details of the bug including a natural language description of a scenario to reproduce the bug. Table 1 is an example bug report (Defects4J Lang-7). In this study, we desired to know whether bug description can help LLM fix bugs by comparing the repair performance of LLMs with and without project information in bug report.

TABLE I
EXAMPLE BUG REPORT (DEFECTS4J LANG-7)

Issue No	LANG-822
Issue title	NumberUtils # createNumber - bad behaviour for leading "–" of BigDecimal, i.e. needs to be moved to createBigDecimal.
Issue description	NumberUtils#createNumber checks for a leading "–" in the string, and returns null if found. This is documented as a work round for a bug in BigDecimal. Returning null is contrary to the ...

- Fault locations. Fault localization is always considered as a prerequisite for bug fixing, hence it is relevant for automated program repair (APR) tools [34].Actually, the repair process of many automated fixing techniques can be divided into three parts: fault localization, patch generation and patch validation [35].In this study, we desired to know whether providing LLM with the location of buggy lines can improve its repair abilities.
- Few-shot prompting. Zero-shot prompting means we directly ask the model to fix the buggy function without giving it any examples of acceptable solutions of similar tasks. Few-shot prompting means that we not only ask the model to fix the buggy function, but also present the model with a few instances of similar tasks, along with their respective inputs and desired outputs. Few-shot learning, which has been studied in many literatures [36]–[39], showed promising results, where they can learn or generalize to a novel category/task based on prior knowledge [40].It aims to help the model adapt to new tasks by providing model with limited number of examples. In this study, we leverage few-shot prompting to see whether it can lead LLM to generate more accurate and coherent output.

We divided the large language model into 32 groups and each group is provided with different information. Then we employed each group's LLM to generate 200 patches for each bug and compared the results of different groups. The first group is given a buggy function and a basic instruction: "Help me fix a buggy function, please directly output a fixed

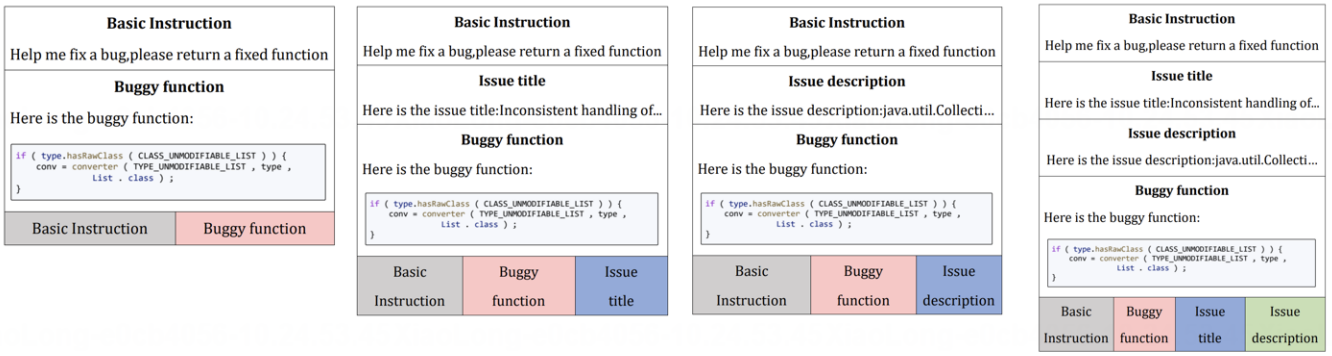

Fig. 3. Prompt settings of group 1 to group 4

function:". The second group is given a buggy function, a basic instruction and an issue title in bug report. The third group is given a buggy function, a basic instruction and an issue description in bug report. The fourth group is given a buggy function, a basic instruction, an issue title and an issue description. Figure 3 shows the 4 different settings. Then fault locations were added to groups 5, 6, 7, and 8 based on the configurations of groups 1, 2, 3, and 4, respectively. We appended comments (/* bug is here */) to all modified lines of each buggy function. Namely, the only difference between group 1 and group 5 is the buggy function. 1-shot prompting is added to group 9 to 16 based on the configurations of group 1 to 8. 2-shot prompting is added to group 17 to 24 based on configurations of group 1 to 8. 3-shot prompting is added to group 25 to 32 based on configurations of group 1 to 8. We randomly combine a buggy program and its fixed version as a bug fix example. For k-shot prompting, we will add k bug fix examples in the prompt. It is guaranteed that the bug fix examples are different from the bug that LLM needs to repair.

C. Patch validation

Finally we integrate each fixed function into the project and call the command-line interface in a python script to test whether the patch passes all the test cases. If there's a patch passing all the test cases among the 200 patches, we conclude that this bug is successfully fixed by LLM.

IV. Experimental Setup

A. Dataset

To evaluate LLM's different repair ability when it is provided with different project information, we choose Defects4J 2.0 [41] as our dataset, which has been widely used in the program repair community. Defects4J is composed of 17 open-source java projects and each projects contains many buggy programs. Each buggy programs in a specific project is accompanied by a bug report and a comprehensive test suite with the associated developer fixes. In addition, Defects4J provides researchers with many command-line interfaces, which enables us to checkout a bug and get information for a specific project. For example, we can get the bug report of a buggy program by calling the command-line interface. Among 835 bugs in the Defects4J 2.0 dataset, we choose bugs which only locate in single-method. Fixes may include multiple lines, but are scoped to a singular function.

B. Metric

If a patch can pass all the test cases from the given test suite of a buggy program, we call this patch a plausible patch. We generate 200 patches for each buggy program under different settings. If there is at least one plausible among these 200 patches, we think this buggy program is successfully fixed by the model. Finally, we count how many bugs can successfully be fixed by model under different settings. The more buggy programs that are successfully repaired, the more capable the model is under this settings.

C. Model

In this study, we choose Codex [42] as our model and we use this model by calling the API provided by openai [43]. Codex, a GPT-3 derived model with 12 billion parameters, is specifically crafted for the purpose of generating code. It is initialized using GPT-3 weights that have been trained on a natural language corpus, and subsequently fine-tuned on an extensive dataset comprising 159 gigabytes of code files. We choose Codex as our model because it is specifically crafted for the purpose of generating code and has shown to achieve impressive performance on code related tasks.

D. Research Questions

We focus on the following research questions :

- **RQ1: How does human-written bug-relevant information affect repair quality?** We explore this by comparing the repair performance of LLMs with and without project information in bug report.
- **RQ2: How does fault location affect repair quality?** In particular, we deeply investigated how buggy lines actually affect model' s repair abilities. We first divided the buggy programs into 2 types. For the first type of program, the model can fixed it when there are fault locations, but model cannot fix it when there are no fault

locations. For the second type of program, the model can fix it when there are no fault locations, but model cannot fix it when there are fault locations. We manually analyzed the features of the two types of bugs.

- **RQ3: How does the value of k in k-Shot prompting affect repair quality?** In this research question, we are dedicated to answering what is the best value of k in k-shot prompting to help LLM fix bugs? Whether the number of examples is always positively correlated with model's repair ability?

V. Results

The result is shown in table 2 and next we will analyse it from 3 perspectives. For simplicity, we refer to "Basic" as A, "Basic +Issue titles" as B, "Basic + Issue description" as C and "Basic +Issue titles+ Issue description" as D in the next analysis.

TABLE II
Overall Results

	Basic	Basic+ Issue titles	Basic+ Issue descriptions	Basic+Issue title +Issue descriptons
K0	174	266	282	302
K1	110	210	215	264
K2	142	256	223	236
K3	165	263	162	168
K0 + FL	225	300	299	312
K1 + FL	184	255	242	276
K2 + FL	211	270	235	235
K3 + FL	210	260	193	185

A. RQ1: Effectiveness of bug descriptions

We compare the number of programs fixed by LLM with and without bug report. To see the result more intuitively, we show the result of group 1 to 4 in Figure 4. We can see that after we add issue titles, the repair ability increase sharply . Under setting K0, model provided with basic instructions fixed 174 buggy programs. Model provided with basic instructions and issue titles fixed 266 buggy programs. Model provided with basic instructions and issue descriptions fixed 282 buggy programs. Moreover, with the provision of basic instructions and issue titles, the model fixed 302 buggy programs. So bug description can definitely help LLM fix bugs. Note that we let the model generate 200 patches for each buggy program, so the result is very convincing.

To understand how fault locations actually help LLM fix bugs, we manually checked the buggy programs that can be fixed with D but cannot be fixed with A. Both of them are under setting K0. we found that without bug-relevant information, it's quite hard for LLM to fix some bugs. For example, Figure 5 shows the buggy program of Math2. If we simply give it to the model, the model will be confused since there are many possible causes of the error. The expression might be wrong or it calls the wrong function or the type of return value is wrong. However, the real cause of the error is integer overflow. The value of "getSampleSize() * getNumberOfSuccesses()" may exceed the range of "int" type

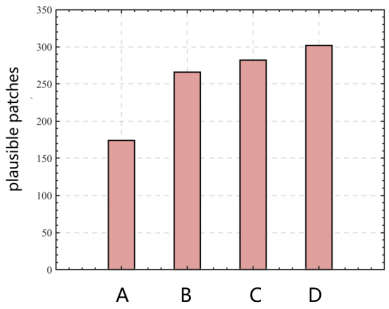

Fig. 4. A refers to basic information, B refers to basic information + issue title, C refers to basic information + issue description and D refers to basic information + issue title + issue description.

and the correct fix is changing the order of operations, which is shown in Figure 6. As shown in Table 3, the bug report

```
public double getNumericalMean() {
return (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize();
}
```

Fig. 5. Math-2

```
public double getNumericalMean() {
return getSampleSize() * (getNumberOfSuccesses() / (double) getPopulationSize());
}
```

Fig. 6. Correct fix of Math-2

of Math-2 points out the cause of the error directly. After we give the bug description, model can fix it immediately. So if we only give the buggy function and basic instructions to the model, the model can only infer cause of the error from the context. Since it has no other available information about the buggy function, it can never fix the buggy function when the function becomes more complicated.

- Finding 1: Bug descriptions can improve repair quality by providing the model with the root cause of the bug.

TABLE III
Bug Report of Math-2

Issue No	MATH-1021
Issue title	HypergeometricDistribution.sample suffers from integer overflow
Issue description	Hi, I have an application which broke when ported from commons math. It looks like the HypergeometricDistribution.sample() method ...

Interestingly ,we found under setting k2 or k3, the repair ability of the model provided with C or D is much worse than the model provided with B. But this doesn't mean bug description is useless since model provided with B is stronger

than model provided with A in all groups. Note that under setting K2 or K3, we will provide much longer prompts for the model. What's worse, few-shot prompting is useless in our findings. In this case, if we continue to add more descriptions, the model will feel more confused since the issue description is much longer than issue title. This phenomenon doesn't occurr in setting K0 or K1 since there isn't so much noisy information. It is indeed that too much irrelevant information will have a negative impact on model's repair ability and we'd better provide the model with concise descriptions.

B. RQ2 : Effectiveness of fault locations

Figure 9 compares the performance of models with and without fault locations. As shown in the figure, the repair capability of model in all group became stronger when it was provided with fault locations. We can see that compared to model which was only provided with basic instruction, the repair abilities of model what was provided with basic instruction and buggy lines increased sharply. As we continue to provide more information, the growth of performance became more gradual. This once again illustrated that there will be a remarkable improvement of model's repair ability when we supply the model with the cause of error on top of providing basic instruction. To understand how fault locations actually help LLM fix bugs, we manually checked the buggy programs which can be fixed with fault locations and cannot be fixed without fault locations(The buggy programs are provided with D). We found that giving the specific fault locations can help the model quickly locate the corresponding location, which is useful for error programs with long contexts, because it can help the model eliminate some confusing information. For example, Figure 7 shows a buggy program. The bug is : "d" is a variable of type double, and assigning "d*d" to a variable of type int will cause a loss of precision. The correct fix is that we should change "int sum = 0" into "double sum = 0", as shown in Figure 8. However, the context of this program is quite long and the figure only shows a part of this

```
<prefix>
int sum = 0;
for(int i = 0;i < pointset.size(); i ++){
    final T p = pointSet.get(i);
    final Cluster<T> nearest = getNearestCluster(resultSet, p);
    final double d = p.distanceFrom(nearest.getCenter());
    sum += d*d;
<suffix>
```

Fig. 7. Math-57

buggy programs. The model only knows the computational expressions are wrong since the return value isn't the same as expected. So it's pretty hard for the model to infer the buggy lines from numerous statements. Based on these discoveries and analyses, we conclude the following finding.

- Finding 2: Marking the error location can help the model accurately pinpoint where modifications are neededwhich is particularly helpful for fixing lengthy buggy codes.

```
<prefix>
double sum = 0;
for(int i = 0;i < pointset.size(); i ++){
    final T p = pointSet.get(i);
    final Cluster<T> nearest = getNearestCluster(resultSet, p);
    final double d = p.distanceFrom(nearest.getCenter());
    sum += d*d;
<suffix>
```

Fig. 8. Correct fix of Math-57

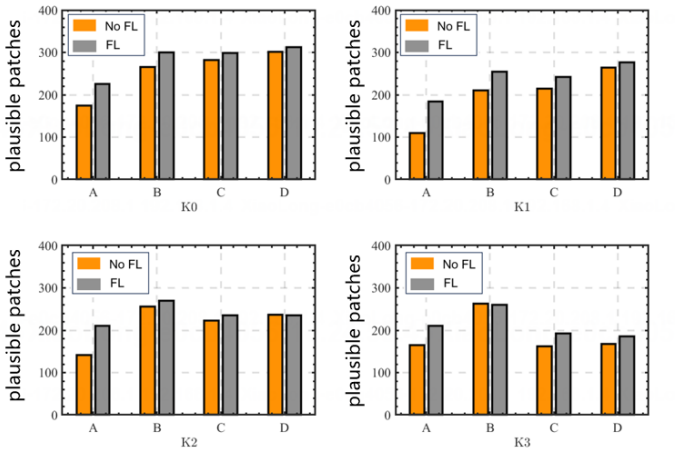

Fig. 9. A refers to basic information, B refers to basic information + issue title, C refers to basic information + issue description and D refers to basic information + issue title + issue description.

Moreover, we found an interesting fact: as shown in Figure 10, there were 47 buggy programs that can be fixed with fault location but can't be fixed without fault locations. Besides, there were 37 buggy programs that can be fixed without fault locations but can't be fixed with fault locations. Note that we let the model generate 200 patches for each buggy program, so we cannot simply attribute this fact to probability. This illustrated that fault locations does not always have a positive impact on LLM-based program repair, which is somewhat different from our intuition.

To figure out the reason behind this phenomenon, we manually analysis the feature of some buggy programs which can be fixed without fault locations but cannot be fixed with fault locations. We list some possible reasons why fault locations are not helpful in some cases and each reason is paired to an example.

1) Fault locations indeed tells LLM the position to modify or infill, but it doesn't tell whether to modify/delete this line or to add a patch of codes. So if we want the model to add a patch of codes, the model can't infer it from the mark(/*bug is here*/) and may modify the codes nearby. What's more, sometimes we can choose multiple places to add this patch ,which means fault locations may misguide LLM in this case. For example, Figure 11 shows a buggy program of Defects4J(Mockito-18).

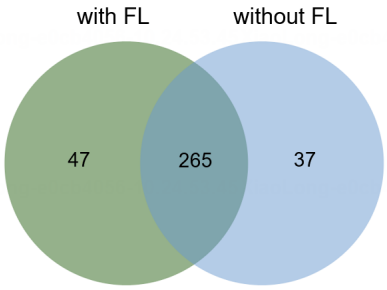

Fig. 10. Venn diagram

To fix this bug, we need to add an ”if” branch in the corresponding position. However, the fault locations are useless since the patch can be added to any positions in the ’if-else’ branch. What’s worse, the added line didn’t appear in the original codes. So the model may think the fault lies in the original codes marked with /*bug is here */. We manually checked the model’s fixes and found most of the fixes modified the codes nearby. For example, a patch changes ”type == Collection.class” into ”type.isAssignableFrom(Collection.class)”. This proves that the model has a wrong understanding of the bug under this settings.

2) Sometimes there are multiple ways to fix a bug, but providing a specific location may let the model focus too much on the buggy lines, which makes the fix monolithic. For example, Figure 12 shows a buggy program of Defects4J (Jsoup-84). The bug of this program lies in the assignment of variable ”el”. If variable ”namespace” is null and variable ”tagName” contains ”:”, el will be assigned to ”doc.createElementNS(””, tagName)”. The correct version is shown in Figure 13. After giving the specific location, the model focused its attention on this buggy line. It may not think of fixing the program with a trinocular operator, because the line to be modified contains too much information. We manually checked the model’s fixes in this case: most of the fixes were focused on this one line and the fixes were wrong. For example, a patch is ”Element el = doc.createElement(tagName);”. Nevertheless, without giving a specific location, the model fix the bug by writing an if statement, which is equivalent to the correct version in Defects4j.

```
<prefix>
-    }else if(type == Iterable.class){
-        return new ArrayList<Object>(0);
     }else if(type == Collection.class){
         return new LinkedList<Object>();
     }else if(type == Set.class){
<suffix>
```

Fig. 11. Mockito-18

3) Among all the buggy programs that can be fixed without FL but cannot be fixed with FL, some of them contains multiple buggy lines and we marked /*bug is here*/ on all modified locations. Nevertheless, the bug locations we mentioned in our study is actually the fixed locations by developers. In the evaluation of fault localization techniques, benchmarks with developer-provided fault locations are often unavailable. In the absence of ground truth fault locations, it is common to use fix locations as substitute fault locations [44]. However, fix location doesn’t equal to fault location and fix location may not reflect the essential reason of the bug in programs. Moreover, sometimes there are multiple ways to fix bugs and it’s not necessary to mark all the buggy lines. So if we manually marked all the places in the buggy function, we are actually providing irrelevant and useless information, which may lower the repair quality according to Finding 2. So we’d better come up with another practical solutions to guide LLM to fix bugs when there are multiple buggy lines instead of clumsily labeling all the fault locations.

Based on these discoveries and analyses, we conclude the following finding.

- Finding 3: Fault locations doesn’t always help LLM fix bugs and may have an negative impact on bug-fixing in some cases.

C. RQ3: Effectiveness of few-shot learning

Table 2 shows the result of each group. To see the result more intuitively, we illustrated the results with two line charts. As shown in Figure 14, the first chart shows the results of groups without fault locations and the second chart shows the results of groups with fault locations. The x-axis of the two charts represents the number of examples in k-shot learning and the y-axis of the two charts represents the number of plausible patches generated by LLM. The color of each line denotes the specific condition we supply with and the condition is marked on the figrure directly. For example, pink represents that we provide the model with basic information, issue title, issue description and fault locations. From the 2

```
<prefix>
String namespace = namespacesStack.peek().get(prefix);
String tagName = sourceEl.tagName();
Element el = doc.createElementNS(namespace, tagName); /*bug is here*/
<suffix>
```

Fig. 12. Jsoup-84

```
<prefix>
String namespace = namespacesStack.peek().get(prefix);
String tagName = sourceEl.tagName();
Element el = namespace == null && tagName.contains(":") ?
  doc.createElementNS("", tagName) : doc.createElementNS(namespace, tagName);
<suffix>
```

Fig. 13. Correct version of Jsoup-84

chart we can see a trend in all groups : the performance of model is the best when we leverage zero-shot learning. As we increase the k in k-shot learning, the result fluctuates irregularly: Results become better for some groups, while for others, results become worse. On top of that, the performance of k-shot learning (k 0) is worse than zero-shot learning in all groups. An intuition is that, although k-shot prompting lets the model know what it is going to do, the model cannot extract specific bug information from it. What's worse, providing irrelevant information may even misguide the model. Prior work [45], [46] has shown that LLMs mostly leverage the few-shot examples to infer the repeated format rather than the task itself in-context. Reynolds [37] also points that example does not always help and the decreased performance was due to semantic contamination from the 1-shot example. Instead of treating the example as a categorical guide, it is inferred that the semantic meaning of the examples are relevant to the task. Namely, if we want LLM to solve problem A by offering it the solution of problem B as an example, we would lower its repair performance. (The type of A and B are different) However, when we use LLM to fix bugs, we can't foresee the type of the bug. So our shots are irrelevant to the actual fix in most cases. Based on all these discoveries and analyses, we conclude the following finding.

- Finding 4: Few-shot learning is not beneficial for enhancing the understanding of LLM regarding the faulty program, and its performance is inferior to zero-shot learning. Consequently, it is advisable to refrain from employing few-shot learning in LLM for addressing APR tasks.

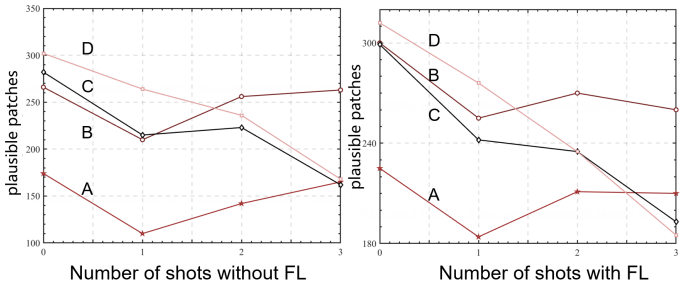

Fig. 14. A refers to basic information, B refers to basic information + issue title, C refers to basic information + issue description and D refers to basic information + issue title + issue description.

VI. Discussion

In this work, we present an extensive study on testing the repair ability of LLM under different settings. However, due to time limitations and a lack of resource, this study is not that perfect because we only test the repair ability of one model (Codex) on one dataset (Defects4J). In the future, we decide to use more different LLMs on more datasets to make our findings more convincing. To take a step further, we can even compare the repair effectiveness of different LLMs and make more useful conclusions by manually analysing more buggy programs of different datasets. There are also many possible ways to further boost the repair ability of LLMs. First, we can try to fine-tune [47] the model for APR tasks. Fine-tuning involves updating the model parameters with a specific training dataset to target a particular downstream task. But this method is quite challenging because not only is fine-tuning expensive as it requires additional model training, but it may also be infeasible in cases where sufficient training datasets are unavailable. Second, in our study, we found that fault locations don't always help and listed some possible reasons. Investigating how to handle these cases is invaluable. On the one hand, we leverage fault locations by directly marking all fixed lines, which may be noisy. On the other hand, it is inappropriate to mark /*bug is here*/ if we want to add a patch of codes. However, in order to fix the buggy code, we need to add code in many cases. Researchers [48] have shown that there are 67.3% patches contain added code and 29.87% patches contain only added code in Defects4J dataset. This illustrates the importance of solving this kind of problem. Besides, fault localization is a prerequisite for APR tools when APR tools are not so powerful. Whether fault localization becomes redundant as APR tools become more powerful? This remains to be answered. Last, we can combine LLM with methods in traditional APR tools [21] such as program analysis. In our study, we only provide the model with a single function and information outside the function is not provided. So in the case when the function is closely related to functions or variables that are outside this buggy function , the model can never fix the buggy function. Program analysis is a promising way to solve this problem. By using program analysis, we can extract the relevant variables and functions and give it to LLM. What's more, our study only discuss buggy programs where errors occurred only within a single function. But bugs are not always scoped to a singular function. It is reported that there are 27% of the patches (107) in Defects4J change more than one method. So this work is of great significance.

VII. Conclusion

In this paper, we conduct an extensive study to figure out how bug descriptions, fault locations and few-shot prompting affect the repair ability of LLM-based APR tools. We use Codex as our model and Defects4J as our dataset. Then we let the model generate 200 patches for each buggy program under 32 different settings. We compared the plausible patches generated by the model under each different settings and manually studied the features of some buggy programs. Eventually we found that bug descriptions are critical for LLM to fix bugs since it will provide the model with the root cause of the bug. However, the description should be as concise as possible since too much irrelevant information will have a negative impact on model's repair ability. What's more, fault locations can help the model fix bugs most of the time since it can help the model accurately pinpoint where the modifications are neededwhich is particularly helpful for fixing lengthy buggy codes. On top of that, fault locations aren't always helpful and may have an negative impact on bug-fixing in some cases. How to better

utilize fault locations is meaningful and still yet to be revealed. Last but not least, we discovered that few-shot learning will not help LLM understand the buggy program and the performance of few-shot prompting is worse than zero-shot learning. So it is better not to apply few-shot learning to LLM when dealing with APR tasks. We hope that our findings will provide some insights for future research.

REFERENCES

[1] Tassey, Gregory. The Economic Impacts of Inadequate Infrastructure for Software Testing[J]. National Institute of Standards & Technology, 2002.

[2] MATTESON S. Report: Software failure caused 1.7 trillion in financial losses in 2017[J]. TechRepublic.[Online]. Available: https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017, 2018.

[3] RICHARDS E. Software's dangerous aspect[J]. The Washington Post, 1990: A25.

[4] BRITTON T, JENG L, CARVER G, et al. Reversible debugging software-quantify the time and cost saved using reversible debuggers[J]. University of Cambridge, 2013.

[5] GAZZOLA L, MICUCCI D, MARIANI L. Automatic software repair: A survey[C]. in: Proceedings of the 40th International Conference on Software Engineering. 2018: 1219-1219.

[6] NGUYEN H D T, QI D, ROYCHOUDHURY A, et al. Semfix: Program repair via semantic analysis[C]. in: 2013 35th International Conference on Software Engineering (ICSE). 2013: 772-781.

[7] MECHTAEV S, YI J, ROYCHOUDHURY A. Directfix: Looking for simple program repairs[C]. in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering: vol. 1. 2015:448-458.

[8] MECHTAEV S, YI J, ROYCHOUDHURY A. Angelix: Scalable multi-line program patch synthesis via symbolic analysis[C]. in: Proceedings of the 38th international conference on software engineering. 2016: 691-701.

[9] LE X B D, CHU D H, LO D, et al. S3: syntax-and semantic-guided repair synthesis via programming by examples[C]. in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 593-604.

[10] LE GOUES C, NGUYEN T, FORREST S, et al. Genprog: A generic method for automatic software repair[J]. IEEE transactions on software engineering, 2011, 38(1): 54-72.

[11] LE X B D, LO D, LE GOUES C. History driven program repair[C]. in: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER): vol. 1. 2016:213-224.

[12] HUA J, ZHANG M, WANG K, et al. Sketchfix: a tool for automated program repair approach using lazy candidate generation[C]. in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.2018: 888-891.

[13] KIM D, NAM J, SONG J, et al. Automatic patch generation learned from human-written patches[C]. in: 2013 35th International Conference on Software Engineering (ICSE). 2013: 802-811.

[14] LIU K, KOYUNCU A, KIM D, et al. TBar: Revisiting template-based automated program repair[C]. in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019: 31-42.

[15] ZHU Q, SUN Z, XIAO Y A, et al. A syntax-guided edit decoder for neural program repair[C]. in:Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 341-353.

[16] JIANG N, LUTELLIER T, TAN L. Cure: Code-aware neural machine translation for automatic program repair[C]. in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021: 1161-1173.

[17] YE H, MARTINEZ M, MONPERRUS M. Neural program repair with execution-based backpropagation[C]. in: Proceedings of the 44th International Conference on Software Engineering. 2022:1506-1518.

[18] WHITE M, TUFANO M, MARTINEZ M, et al. Sorting and transforming program repair ingredients via deep learning code similarities[C]. in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2019: 479-490.

[19] LE GOUES C, DEWEY-VOGT M, FORREST S, et al. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 dollar each[C]. in: 2012 34th International Conference on Software Engineering (ICSE). 2012: 3-13.

[20] XIA C S, WEI Y, ZHANG L. Automated program repair in the era of large pre-trained language models[C]. in: Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery. 2023.

[21] LIU K, KOYUNCU A, KIM D, et al. Avatar: Fixing semantic bugs with fix patterns of static anal- ysis violations[C]. in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2019: 1-12.

[22] VASWANI A, SHAZEER N, PARMAR N, et al. Attention Is All You Need[J]. arXiv, 2017.

[23] LIU Y. Fine-tune BERT for Extractive Summarization[J]. 2019.

[24] YANG Z, DAI Z, YANG Y, et al. XLNet: Generalized Autoregressive Pretraining for Language Understanding[J]. 2019.

[25] SUTSKEVER I, VINYALS O, LE Q V. Sequence to Sequence Learning with Neural Networks[J]. Advances in neural information processing systems, 2014.

[26] DEVLIN J, CHANG M W, LEE K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding[J]. 2018.

[27] RADFORD A, WU J, CHILD R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8): 9.

[28] RAFFEL C, SHAZEER N, ROBERTS A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. The Journal of Machine Learning Research, 2020, 21(1): 5485-5551.

[29] XIA C S, ZHANG L. Less training, more repairing please: revisiting automated program repair via zero-shot learning[C]. in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022: 959-971.

[30] XIA C S, DING Y, ZHANG L. Revisiting the Plastic Surgery Hypothesis via Large Language Models[J]. arXiv preprint arXiv:2303.10494, 2023.

[31] XIA C S, ZHANG L. Keep the Conversation Going: Fixing 162 out of 337 bugs for 0.42 dollar each using ChatGPT[J]. arXiv preprint arXiv:2304.00385, 2023.

[32] FAKHOURY S, CHAKRABORTY S, MUSUVATHI M, et al. Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions[J]. arXiv preprint arXiv:2304.03816, 2023.

[33] ZHANG T, JIANG H, LUO X, et al. A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions[J]. Computer Journal, 2018, 59(5): 741-773.

[34] SOREMEKUN E, KIRSCHNER L, BÖHME M, et al. Evaluating the Impact of Experimental Assumptions in Automated Fault Localization[C]. in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 2023: 159-171. DOI: 10.1109/ICSE48619.2023.00025.

[35] JIANG J, CHEN J, XIONG Y. Survey of Automatic Program Repair Techniques [J][J]. Journal of Software, 2021, 32(9): 2665-2690.

[36] GARCIA V, BRUNA J. Few-Shot Learning with Graph Neural Networks[J]. 2017.

[37] REYNOLDS L, MCDONELL K. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm[J]. 2021.

[38] BROWN T B, MANN B, RYDER N, et al. Language Models are Few-Shot Learners[J]. 2020.

[39] LI F F, FERGUS R, PERONA P. A Bayesian Approach to Unsupervised One-Shot Learning of Object Categories[C]. in: IEEE International Conference on Computer Vision. 2003.

[40] LU J, GONG P, YE J, et al. Learning from Very Few Samples: A Survey[J]. 2020.

[41] JUST R, JALALI D, ERNST M D. Defects4J: a database of existing faults to enable controlled testing studies for Java programs[J]. 2014.

[42] Evaluating Large Language Models Trained on Code[J]. 2021.

[43] Openapi. https://openai.com/api, 2022.

[44] SOREMEKUN E, KIRSCHNER L, BÖHME M, et al. Evaluating the impact of experimental assumptions in automated fault localization[C]. in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 2023: 159-171.

[45] WORK W M I C L. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?[J].

[46] KOJIMA T, GU S S, REID M, et al. Large language models are zero-shot reasoners[J]. Advances in neural information processing systems, 2022, 35: 22199 to 22213.

[47] RADFORD A, NARASIMHAN K, SALIMANS T, et al. Improving language understanding by generative pre-training[J]. 2018.
[48] SOBREIRA V, DURIEUX T, MADEIRAL F, et al. Dissection of a bug dataset: Anatomy of 395 patches from defects4j[C]. in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2018: 130-140.