The Fact Selection Problem in LLM-Based Program Repair

Nikhil Parasaram
University College London
nikhil.parasaram.19@ucl.ac.uk

Huijie Yan[†]
University College London
huijie.yan.20@ucl.ac.uk

Boyu Yang[†]
University College London
boyu.yang.21@ucl.ac.uk

Zineb Flahy
University College London
zineb.flahy.21@ucl.ac.uk

Abriele Qudsi
University College London
abriele.qudsi.21@ucl.ac.uk

Damian Ziaber
University College London
damian.ziaber.21@ucl.ac.uk

Earl T. Barr
University College London
e.barr@ucl.ac.uk

Sergey Mechtaev[*]
Peking University
mechtaev@pku.edu.cn

Abstract—Recent research has shown that incorporating bug-related facts, such as stack traces and GitHub issues, into prompts enhances the bug-fixing capabilities of large language models (LLMs). Considering the ever-increasing context window of these models, a critical question arises: what and how many facts should be included in prompts to maximise the chance of correctly fixing bugs? To answer this question, we conducted a large-scale study, employing over 19K prompts featuring various combinations of seven diverse facts to rectify 314 bugs from open-source Python projects within the BugsInPy benchmark. Our findings revealed that each fact, ranging from simple syntactic details like code context to semantic information previously unexplored in the context of LLMs such as angelic values, is beneficial. Specifically, each fact aids in fixing some bugs that would remain unresolved or only be fixed with a low success rate without it. Importantly, we discovered that the effectiveness of program repair prompts is non-monotonic over the number of used facts; using too many facts leads to subpar outcomes. These insights led us to define the fact selection problem: determining the optimal set of facts for inclusion in a prompt to maximise LLM's performance on a given task instance. We found that there is no one-size-fits-all set of facts for bug repair. Therefore, we developed a basic statistical model, named MANIPLE, which selects facts specific to a given bug to include in the prompt. This model significantly surpasses the performance of the best generic fact set. To underscore the significance of the fact selection problem, we benchmarked MANIPLE against the state-of-the-art zero-shot, non-conversational LLM-based bug repair methods. On our testing dataset of 157 bugs, MANIPLE repairs 88 bugs, 17% above the best configuration.

Index Terms—automated program repair, large language models, prompt engineering

I. INTRODUCTION

When debugging and fixing software bugs, developers seek bug-related information from a diverse array of sources. Such sources include the buggy code's context, documentation, error messages, outputs from program analysis, etc. Individual pieces of this information, which following recent work we refer to as *facts* [1], have been demonstrated by previous studies to enhance LLMs' bug-fixing efficacy when incorporated into the prompts [2], [3], [4]. Given the ever-increasing context window

of cutting-edge LLMs, a critical question emerges: "Which specific facts, and in what quantity, should be integrated into the prompt to optimise the chance of correctly fixing a bug?"

This work is a systematic effort to investigate how to construct effective prompts for LLM-based automated program repair (APR) by composing facts extracted from the buggy program and external sources. We identified seven facts: those individually studied in the context of APR by previous work, such as the buggy code's context [5], [2], [6], GitHub issues [3], and stack traces [7]; Angelic values, a semantic fact previously unexplored in the context of LLM-based APR, but that has been successfully used for debugging [8] and repair [9]; and those chosen based on our intuition as developers. Our study was conducted on 314 bugs in open source Python projects from the BugsInPy [10] benchmark.

Our first experiment aims to confirm the utility of the considered facts. Specifically, for each fact, if the potential inclusion of this fact in APR prompts helps to repair some additional bugs, or increases the probability of fixing some bugs. To answer this question, we constructed over 19K prompts tasked to repair the buggy function and containing different subsets of the seven facts for the 314 bugs. Then, we queried an LLM to generate patches, and evaluated the patches using the provided test suites. Finding 1 confirms the utility of each fact, that is each fact helped repair at least one bug that was not repaired by any prompt without this fact. Moreover, all the facts have statistically significant positive impact on the probability of fixing a bug in a single attempt.

Given the utility of each fact, it is tempting to assume that adding more facts always enhances LLM's performance. Contrary to this intuition, Finding 2 reveals that APR prompts are non-monotonic over facts: adding more facts may degrade LLM's performance. An experiment involving 157 bugs showed that prompts incorporating all available facts resulted in 12 fewer bug fixes and exhibited an 8.2% lower probability of repairing a bug within a single attempt compared to the most effective subset. This finding is non-obvious, because each fact may contain crucial information for fixing the bug, and although previous research showed that LLMs do not robustly make use of information in long input contexts [11],

---

[†]These authors contributed equally to this work.
[*]Corresponding author.

```
Please fix the buggy function provided below and output a
    ↪ corrected version.
<.. CoT instructions are omitted ...>

## The source code of the buggy function
```python
# this is the buggy function you need to fix
def read_json(
<... a part of code is omitted ...>
    return result
```
A test function that the buggy function fails
```python
def test_readjson_unicode(monkeypatch):
<... a part of code is omitted ...>
        tm.assert_frame_equal(result, expected)
```
The error message from the failing test
```text
monkeypatch = <_pytest.monkeypatch.MonkeyPatch object at 0
    ↪ x7f567d325d00>
<... a part of message is omitted ...>
pandas/_libs/testing.pyx:174: AssertionError
```
Runtime values and types of variables inside the buggy
    ↪ function
compression, value: '‘infer’', type: ‘str’
<... some variables are omitted ...>
lines, value: ‘False’, type: ‘bool’

Expected values and types of variables during the
    ↪ failing test execution
path_or_buf, expected value: '‘/tmp/tmphu0tx4qstest.json’',
    ↪ type: ‘str’
<... some variables are omitted ...>
result, expected value: <...omitted...>, type: ‘DataFrame‘

A GitHub issue for this bug
```text
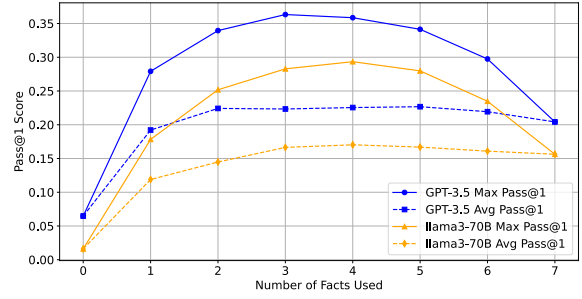Code Sample, a copy-pastable example if possible
<... a part of text is omitted ...>
However, when read_json() is called without encoding
    parameter, it calls built-in open() method to open a
    ↪ file and open() uses return value of locale.
    ↪ getpreferredencoding() to determine the encoding
    ↪ which can be something not utf-8
```
```

Fig. 1: A simplified APR prompt incorporating various facts for fixing pandas:128. ... shows information omitted for brevity. ... shows a part of the GitHub issue that is essential to correctly fix the bug. Too much information in the prompt "distracts" the LLM from the relevant part of the issue description, significantly reducing pass@1 and the correctness rate.

and their performance dramatically decreases when irrelevant information is included in a prompt [12], the trade-off between utility of information and the ability of LLM to process long contexts has not investigated.

The non-monotonicity of APR prompts and, simultaneously, the utility of each fact led us to define *the fact selection problem*: determining the optimal set of facts for inclusion in prompts to maximise LLM's performance on given tasks. It can be viewed as a variant of feature selection of classical machine learning for LLM prompt engineering. We consider two instances of fact selection: *universal fact selection* when the selected facts do not depend on a specific task instance, *i.e.* the bug, and *bug-tailored fact selection* when the fact set is bug-specific.

If there was a universal set of facts that is effective for all bugs, it would significantly simplify the development of LLM-based APR tools. However, our experiments showed that universal fact selection is suboptimal compared to bug-tailored

Fig. 2: Comparison of pass@1 (the vertical axis) for around 10K prompts incorporating subsets of the seven considered facts (the horizontal axis) computed over 15 responses for repairing 157 Python bugs within the BugsInPy benchmark. Zero facts corresponds to the prompt containing only the buggy function without any additional information about the bug. The graph plots the average pass@1 score (dashed lines) and the maximum pass@1 score (solid lines) across all bugs. This graph clearly shows the non-monotonic nature of APR prompts over facts for GPT-3.5 and Llama3-70B.

fact selection. Specifically, Finding 3 identified that there is no single subset of the seven considered facts that is sufficiently effective across all bugs in the dataset. Meanwhile, enumerating sets of facts via *e.g.* greedy strategies while repairing each bug might be impractical, because of the high cost of LLM queries and the necessity to generate multiple responses due to LLMs' nondeterminism. As a practical compromise, we trained a statistical model that we call MANIPLE. It is designed to select facts contingent upon the features of a specific bug. Empirical evidence shows that MANIPLE significantly outperforms a universal fact selection strategy, fixing 11 more bugs than a generic set of facts that exhibited the optimal performance on the training data.

We benchmarked MANIPLE against state-of-the-art zero-shot non-conversational LLM-based APR techniques. On our testing set, MANIPLE repaired 17% more bugs, highlighting the practical impact of the fact selection problem.

The contributions of this work are:

- A large scale systematic study of a diverse set of bug-pertinent facts for APR prompts.
- An empirical evidence of the utility of each considered fact for program repair, including angelic values, previously unexplored in the context of LLMs.
- An empirical evidence of the non-monotonicity of APR prompts over bug-related facts, *i.e.* adding more bug-related facts into APR prompts may degrade LLM's bug-fixing performance.
- A motivation and introduction of the fact selection problem for LLM-based APR.
- MANIPLE, a bug-tailored fact selection model that, for a given bug, chooses a set of fact to construct an effective APR prompt; This model significantly outperforms previous related techniques.

All code, scripts, and data necessary to reproduce this work are available at https://github.com/PyRepair/maniple.

II. MOTIVATING EXAMPLE

To illustrate the importance of the fact selection problem, we consider the bug pandas:128 [13] in the Pandas data analysis library within the BugsInPy benchmark. This bug arises due to incorrectly handling the default encoding in the Pandas' `read_json` function. The developer patch for this bug involves setting the encoding to UTF-8 when it is not specified by adding the following lines to the buggy function:

```
+       if encoding is None:
+           encoding = "utf-8"
```

When GPT-3.5-Turbo [14] was prompted to rectify the bug solely based on the source code of the buggy function, it did not successfully address the issue in any of 15 responses. A likely explanation of this failure is that the function itself does not contain any inherently incorrect code, so fixing this bug requires external information.

Drawing upon existing literature on LLM-based APR and relying on our intuition as developers, we assembled a diverse set of bug-related facts to incorporate into the prompt. These include the buggy function's context, the failing test case, the error message, the runtime values of local variables, their angelic values (values that, when taken by program variables during test execution, result in successful passage of the test), and the GitHub issue description. Figure 1 gives a simplified representation of the resulting prompt. Adding these facts enabled the LLM to generate a plausible patch, *i.e.* a patch that passes the tests, in four out of 15 responses. However, only two of these patches were correct. The other two hard-coded UTF-8 as the only encoding instead of the default one. The causes of failures to fix the bug included "forgetting" to change the function despite correct chain-of-thought reasoning [15], or hard-coding the encoding inconsistently.

Interestingly, when we removed all facts but the code of the buggy function, the runtime variable values and the GitHub issue, it significantly raised the success rate. Specifically, the LLM generated plausible patches in 12 out of 15 responses, and 11 out of these 12 were correct. A similar high success rate was demonstrated by the prompt with only the buggy function and the GitHub issue. We posit that this is because redundant or irrelevant information in the original prompt "distracted" the LLM from critical details in the GitHub issue (highlighted in Figure 1) necessary to repair the bug [12].

We refer to this phenomenon, that adding more facts may degrade LLM's performance, as the *non-monotonicity* of prompts over facts. To provide stronger empirical evidence, we conducted a large-scale experiment with 19K prompts containing various subsets of seven facts on 314 bugs in Python projects. Figure 2 shows how pass@1, which estimates the probability of generating a plausible patch in a single trial, depends on the number of included facts. The graph shows two types of lines: the solid line corresponds to the scenario when we select the most effective combination of facts for each bug. The dashed line indicates the performance of an average fact set. It is evident that both these functions are non-monotonic for both GPT-3.5 and Llama3-70B. This phenomenon of non-monotonicity and its impact on prompt performance are discussed in more detail in Section IV-B.

Apart from the non-monotonicity of prompts, we also discovered that each of the considered facts helps to fix some bugs that cannot be fixed without it. These observations motivated us to formulate the *fact selection problem*, the problem of selecting facts for a given bug to maximise the chance of repairing it and propose a model MANIPLE that selects effective facts based on features of a given bug.
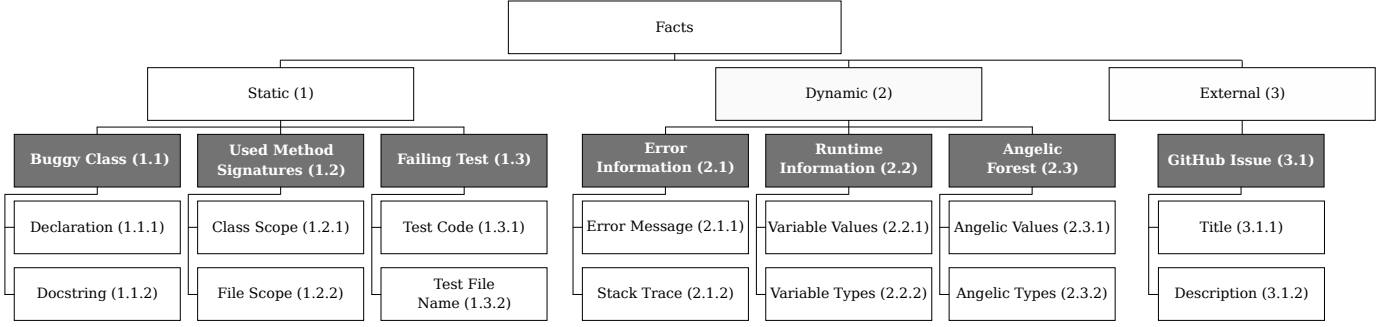
III. STUDY DESIGN

This section discusses the experimental setup, the facts chosen for our study, and how they are represented in prompts.

A. Experimental Setup

We denote the set of bugs as B and the set of all bug-relevant facts as $\mathbb{F}$. Since we aim to investigate how using various facts impacts the success of APR, we define the set of all jobs as $\mathbb{J} = B \times 2^{\mathbb{F}}$, which pairs bugs with sets of facts.

Zero-shot Prompting for APR: A advantage of large language models, such as ChatGPT [14], is they can be adapted to a downstream task without retraining via *prompt engineering* [16]. A prompt refers to the input or instruction given to the model to elicit a response. Prompting typically takes either the form of *zero-shot*, *i.e.* directly providing the model a task's input, or *few-shot* [17], where the model is provided with a few examples. In this work, we investigate a zero-shot APR approach. Although the few-shot approach is promising, it requires finding high-quality examples [18], which we leave for future work.

Function-granular Perfect Fault Localisation: APR tools repair bugs by first localising suspicious locations. For an objective evaluation of APR tools, Liu *et al.* [19] argues for the use of *perfect fault localisation (PFL)*, that is, when the buggy locations are known to the tool. PFL can resolve to difference granularity levels: notably, the line or the function. Liu *et al.* argues that fault localisation tools do not offer the same accuracy in identifying faulty locations at different granularities, "making function-level granularity appealing for limiting unnecessary responses on fault positive locations". We found that, in BugsInPy [10], the average Ochiai [20] rank of the buggy line is 2502 and the buggy function is 22 across the entire codebase, making it much more likely to localise the buggy function than the buggy line. Apart from that, although 70% of the bugs in BugsInPy require modifying only a single function, 65% of them modify multiple lines within this function. Localising a bug to multiple lines is harder than targeting individual lines. Meanwhile, cutting-edge models, like GPT-3.5-Turbo, effectively fix bugs even without specifying the exact lines, *i.e.* when only the buggy function is provided. Consequently, and in contrast to some previous studies [21], [22], [2], this paper adopts function-granular PFL as the standard approach for evaluating APR tools. We implement this by providing the buggy function as context in the prompts for the LLM, enabling it to focus on fixing issues within the specified function.

Fig. 3: This work uses seven facts (dark rectangles) across three categories for constructing program repair prompts. Each fact is composed of related pieces of information. Each prompt contains the buggy function to be repaired; the facts can be included based on the employed fact selection strategy.

Python Bug Benchmark: APR tools are typically compared on datasets of bugs extracted from real world projects, such as Defects4J [23] for Java and BugsInPy [10] for Python. In this work, we use BugsInPy because of Python's ever-increasing importance and popularity. BugsInPy contains 501 bugs from 17 popular Python projects such as Pandas [24] and Matplotlib [25]. Among them, we selected a subset of 314 bugs, which we refer to as **BGP314**, that require modifications within a single function, due to our PFL approach, and that we were able to reproduce. To investigate APR performance on various classes of defects, we consider three parts of BGP314:

- **BGP157PLY1**: This dataset comprises 157 bugs, which have been uniformly selected from BGP314 for the purpose of training and analysis.
- **BGP157PLY2**: Consisting of 157 bugs, this dataset is the complement of BGP157PLY1 in BGP314. Used for evaluating fact selection strategies.
- **BGP32**: A subset of 32 bugs uniformly sampled from BGP157PLY1, intended for preliminary studies on finding parameters, such as determining the right response count to reduce the variance.

LLM Nondeterminism: Nondeterminism in LLMs leads to varying outcomes between responses, which poses a challenge for analysing results [26]. To alleviate it, we use the pass@k measure, which represents the probability that at least one query out of k succeeds at solving a problem. Previous work [27] recommends estimating pass@k as

$$\text{pass@k}(LLM(Q)) \triangleq \mathbb{E}_Q \left[1 - \frac{\binom{n-C}{k}}{\binom{n}{k}} \right] \quad (1)$$

where $\mathbb{E}_J$ denotes expectation over the set of LLM responses to the set of queries (prompts) Q, n is the number of responses obtained from the LLM, where $n > k$ and C is the number of successes found in the n responses. Our task is program repair, so we deem a response successful if the extracted patch satisfies a correctness criterion, which we approximate with passing a test suite. A pilot study using BGP32 reveals that when $n = 15$ and $k = 1$, pass@k exhibits the average standard deviation of *ca.* 0.04, and that further increasing n only marginally decreases standard deviation. For more details,

refer to Appendix A in the full version of this paper on arXiv [28]).

For generate-and-validate APR [29] that iteratively generates and tests patches until it finds one that passes, a commonly used measure is the number of bugs for which at least one patch passes the tests among LLM's responses:

$$\#\text{fixed}(LLM(J)) \triangleq \left| \{b \mid j \in J, C_j > 0\} \right| \quad (2)$$

where $j = (b, F)$ is a prompt, and C_j is the number of responses that pass the tests for the prompt j.

Test-overfitting in Program Repair: APR techniques repair bugs *w.r.t.* correctness criteria, such as tests or formal specification. Since tests do not fully capture the intended behaviour, automatically generated patches based on tests may be incorrect [30]. Thus, the APR literature distinguishes between *plausible* patches, patches that pass the tests, and *correct* patches, patches that satisfy the intended requirements. Since manually labelling a large number of patches is resource-intensive and error-prone, most analyses of the fact selection problem with pass@k and #fixed in this paper count plausible patches as successes. We only label correct patches when comparing our tool with other APR techniques in Section VI.

In our experiments, we utilised the latest version of GPT-3.5-Turbo, specifically gpt-3.5-turbo-0125, which features a 16K context window. For our studies, GPT-3.5-Turbo was run with $n = 15$ responses on BGP157PLY1, BGP157PLY2, and with $n = 30$ for parameter exploration on BGP32. Additionally, we used llama3-70B, which offers an 8K context window and was run exclusively on BGP157PLY1. As of March 2024, the cost of reproducing the experiments detailed in this paper, using the OpenAI API [31] for GPT-3.5 and DeepInfra [32] for llama3-70B, is estimated at \$479 + \$269.

B. Bug-Related Facts

The considered facts were collected from previous LLM-based APR research, previous non-LLM-based APR literature, and our intuition as developers. We conducted a pilot study on BGP32 to validate fact utility. In total, we collected 14 pieces of information, but since many of them are related, we grouped them into seven facts, which we refer to as $\mathbb{F}$. These seven facts are divided into three categories: static, dynamic, and external, as shown in Figure 3.

Buggy Class (1.1): The declaration of a class containing the buggy function provides a broader context and dependencies. A class docstring offers insight into the overall purpose and functionality of the class.

Used Method Signatures (1.2): Considering methods used within the buggy function, as shown by Chen *et al.* [6], allows for the analysis of dependencies and potential side effects that might contribute to the incorrect behaviour.

Failing Test (1.3): The code of a failing test, as shown by Xia *et al.* [4] provides useful context for repairing a buggy function as it specifically highlights the conditions under which the program fails.

Error Information (2.1): Previous approaches showed that using error messages [4] and stack traces [7] improves LLM's bug-fixing performance.

Runtime Information (2.2): Runtime values and types of the function's parameters and local variables during the failing test execution provide an LLM with concrete data about the program's behaviour.

Angelic Forest (2.3): For a given program location, a variable's *angelic value* [8] is a value that, if bound to the variable during the execution of a failing test, would enable the program to pass the test. Angelic forest [9], previously applied for synthesis-based repair, is a specification for a program fragment in the form of pairs of initial states and output angelic values, such that if the fragment satisfies these pairs, then the program passes the test.

Inspired by this approach, we added a variant of angelic forest to a prompt; this variant combines variable values at the beginning of a function's execution coupled with the angelic values at the end of a function's execution, *i.e.* the input/output requirements of the function. Since Python is dynamically typed, we specify both the values and types of variables. Angelic values can be computed using symbolic execution [9], [8]; however, due to the immaturity of Python symbolic execution engines, we were unable to execute them on bugs in BugsInPy. Thus, we extracted angelic values from the correct versions of the programs via instrumentation.

GitHub Issue (3.1): A GitHub issue, when available, provides important contextual information for fixing the bug, as shown by Fakhoury *et al.* [3].

To denote subsets of $\mathbb{F}$, we utilise seven-width *bitvectors*, where the i-th bit indicates whether the i-th fact in our taxonomy (Figure 3) is included in the set. For example, 0000100 corresponds to the set containing only the runtime information (2.2).

C. Prompt Design

We construct prompts via the prompt engineer $E\colon B \times 2^{\mathbb{F}} \to \Sigma^*$, which builds a prompt over the alphabet Σ to repair an input bug using a subset of facts from $\mathbb{F}$. The prompt is constructed with the directive "Please fix the buggy function provided below and output a corrected version" along with the included subset of facts. The buggy function's code, together with its docstring, is provided as part of the prompt for the LLM to effectively fulfill this directive. Each fact is incorporated via a specialised prompt template. Figure 1 shows an example prompt with incorporated facts, and the fact templates are detailed in Appendix B of the full version of this paper on arXiv [28]).

We employed the standard chain-of-thought [15] approach by instructing the LLM to reason about the provided facts, as detailed in Appendix C of the full version of this paper on arXiv [28].

In our preliminary experiments, we discovered that LLM often generates incorrect import statements, which makes it hard to automatically extract patches from the responses and apply them to the code. To address it, we explicitly added the import statements in the current file to the prompts. A small study showed that this consistently improves the success rate, as detailed in Appendix D of the full version of this paper on arXiv [28].

IV. THE FACT SELECTION PROBLEM

Let an LLM be a function from a string, *i.e.* a prompt, to a set of strings, the responses R. We consider an arbitrary measure $m\colon 2^R \times 2^C \to \mathbb{R}$ that scores a set of LLM responses *w.r.t.* some correctness criteria C that maps correct patches to a high score and incorrect patches to a low score, and a prompt engineer $E\colon B \times 2^{\mathbb{F}} \to \Sigma^*$.

Definition IV.1 (Fact Selection Problem). Given a set of bug-relevant facts $\mathbb{F}$, a prompt engineer E, a buggy program b, and correctness criteria for that buggy program C_b, the *fact selection problem* is to find $F \subseteq \mathbb{F}$ that maximises

$$\arg\max_{F \in 2^{\mathbb{F}}} m(\text{LLM}(E(b, F)), C_b) \qquad (3)$$

C_b encompasses any of the standard correctness criteria such as a test suite or a specification, *etc.* $\overset{*}{F}_b$ denotes an optimal solution of Equation (3) for b; We use $\overset{*}{F}_b(\mathbb{F})$ to denote the optimal F for the buggy program b over the fact set $\mathbb{F}$.

Similarly, we use $\overset{*}{F}_B(\mathbb{F})$ to denote the optimal fact set F over all the buggy programs $b \in B$ and the fact set $\mathbb{F}$. This can be defined as the solution to the equation below.

$$\arg\max_{F \in 2^{\mathbb{F}}} \sum_{b \in B} m(\text{LLM}(E(b, F)), C_b)$$

We aim to answer the following questions in this section:

- *How does the inclusion of each fact affect the overall effectiveness of a program repair prompt?*
- *Is there point beyond which adding facts to a program repair prompt degrades its performance?*
- *Can a fixed subset of facts be universally optimal up to a tolerance of ϵ for bug resolution across various bug sets?*

The first question examines the impact of each fact on the repair effectiveness, questioning whether every fact contributes positively to the resolution process. Section IV-A answers this question affirmatively, showing the inherent value of each fact. The second question delves into the potential for diminishing returns or even detrimental effects from overloading a prompt with too many facts, suggesting an optimal threshold for fact

Fact	GPT-3.5		Llama3-70B	
	Gain	**Shapley**	**Gain**	**Shapley**
Error Info.	0.48	0.54	0.41	0.32
GitHub Issue	0.44	0.51	0.20	0.17
Angelic Forest	0.08	0.11	0.36	0.28
Failing Test	0.06	0.08	0.17	0.15
Runtime Info.	0.03	0.05	0.08	0.07
Buggy Class	-0.03	-0.05	0.03	0.03
Used Method S.	-0.12	-0.18	0.01	0.00

TABLE I: We report Gain (Equation (5)) and Shapley values (scaled by 16) for uniform fact selection on BGP157PLY1. Gain quantifies the average percentage increase in prompt repair performance from the fact.

inclusion that maximises prompt efficacy. If there is no such point, then the optimal strategy will be to include all the facts. Section IV-B shows that, on our dataset, adding facts to a prompt is non-monotonic. Formally, the final question asks whether, $\forall b \in B$, the following equation holds:

$$m\big(\text{LLM}(E(b, \overset{*}{F}_B)), C_b\big) = m\big(\text{LLM}(E(b, \overset{*}{F}_b)), C_b\big) + \epsilon \quad (4)$$

This equation asks whether one can select a fact set for a set of bugs that is as effective as a fact set tailored to each bug. Section IV-C answers this question by showing that this statement does not hold. The above answers, combined, establish the importance of the fact selection problem.

A. Fact Utility

A fact should only be considered for inclusion in a prompt if it has a potential to improve the outcome. To confirm the utility of the considered facts $\mathbb{F}$, we simplify the premise by assuming that facts are independent and pose two questions: "What is the utility of each individual fact in improving repair performance on our dataset if we select the most effective fact set for each bug?" and "What is the utility of each individual fact in improving repair performance on our dataset if we select a random fact set for each bug?" The first question addresses the potential effectiveness when we precisely know which facts to choose for a specific bug. This notion of utility, which we refer to as *utility under optimal fact selection*, is relevant when we have a method to closely approximate an optimal solution $\overset{*}{F}_b(\mathbb{F})$ to Equation (3), *i.e.* choose the most effective facts for each bug $b \in B$. The second question explores the expected outcomes when we lack specific knowledge about which facts to select, and hence make a random choice. This notion of utility, which we call *utility under uniform fact selection*, is relevant when solving Equation (3) is either difficult or infeasible.

To estimate the utility of each fact, we generated prompts containing all subsets of the considered 7 facts (the remaining one, the buggy function, is always present in the prompt), which resulted in a total of 19228 prompts for BGP314, which is less than 314×2^7 since some facts are not available for some bugs. For each prompt, we computed 15 responses to estimate the measures pass@1 and #fixed. This enabled us to both evaluate

an optimal selection strategy by explicitly considering $\overset{*}{F}_b$ for each bug, and a uniform selection strategy.

We evaluate the utility of individual facts under uniform selection using two complementary measures: Shapley [33] and a new measure we introduce and call *fact gain*, defined in Equation (5). We report fact gain along with Shapley for two reasons: (1) Shapley's results are hard to interpret and (2) we have the luxury of exhaustive enumeration, since our fact set is small. Fact gain computes the net increase in pass@1 scores due to the addition of a specific fact f; we defined it by adapting relative change [34] to our problem domain by setting the reference value to fact subsets that do not contain the measured fact f. Let $J_f = \{(b, F) \in J \mid f \in F\}$ be the subset of prompts whose fact set F includes the fact f, and $J_{\overline{f}} = \{(b, F) \in J \mid f \notin F\}$ be those prompts that do not. Let $R_f = LLM(J_f)$ and $R_{\overline{f}} = LLM(J_{\overline{f}})$. Then the gain of each fact is:

$$A(f) = \frac{\text{pass@k}(R_f) - \text{pass@k}(R_{\overline{f}})}{\text{pass@k}(J_{\overline{f}})} \quad (5)$$

$A(f)$ computes the change in the likelihood of generating a successful repair when the fact f can be used in a prompt.

Table I showcases the significance of each fact in APR prompts, leveraging both aggregate gain ($A(f)$) and Shapley values, as defined in Equation (5). Another interesting observation from the experimental results is that the facts "Buggy Class" and "Used Method" Signatures exhibit a negative aggregate gain for GPT-3.5, indicating that their inclusion might adversely affect the repair outcome on average. For Llama3-70B, these facts have an aggregate gain close to zero, still making them the least beneficial among the considered facts. Nonetheless, each of these facts allows GPT-3.5 to fix 4 additional bugs that were not fixed without them. For Llama3-70B, "Buggy Class" enables fixing 1 additional bug, while "Used Method Signatures" contributes to fixing 5 additional bugs. This shows the importance of fact selection.

To demonstrate the utility of facts $\mathbb{F}$ under optimal fact selection, we compute the number of bugs that were fixed exclusively when each fact was available. These exclusive fixes are reported in the column "# Excl." for both GPT-3.5 and Llama3-70B in Table II. This table highlights the unique bug-fixing contribution of each fact, showing the distinct set of bugs resolved by each fact across the two models. It further shows that all facts improve performance. These improvements are statistically significant for both models, with no p-value exceeding 0.004.

Second, we analysed each fact's utility under optimal selection by how its inclusion in or its exclusion affects pass@k. We do so by simulating the scenario where specific facts are missing: If a fact f were missing, we would be forced to compute $\overset{*}{F}_b$ over $\mathbb{F} - \{f\}$ for each bug $b \in B$. The baseline for this scenario is when all the facts are available.

Each row in the table in Table II details the pass@1 attainable by the best prompts, with and without a specific fact (denoted by f). For each bug $b \in B$, prompts are constructed using optimal

Fact f	GPT-3.5			Llama3-70B		
	# Excl.	$\overset{*}{F}_b(\mathbb{F} - \{f\})$	Δ	# Excl.	$\overset{*}{F}_b(\mathbb{F} - \{f\})$	Δ
Error Info.	9	0.331	0.071	8	0.257	0.0493
Failing Test	7	0.375	0.027	7	0.281	0.0251
Angelic Forest	7	0.371	0.032	5	0.245	0.0607
Buggy Class	4	0.392	0.010	1	0.290	0.0166
Used Method S.	4	0.393	0.009	5	0.285	0.0208
GitHub Issue	3	0.341	0.062	6	0.263	0.0429
Runtime Info.	2	0.384	0.019	8	0.288	0.0183

TABLE II: # Excl. shows the bugs that could only be fixed by including the specific fact under optimal fact selection for BGP157PLY1 . $\overset{*}{F}_b(\mathbb{F} - \{f\})$ shows performance of the best facts when **f** cannot be selected. Δ represents the drop in performance due to excluding the fact from the bug's best performing fact set.

fact sets $\overset{*}{F}_b(\mathbb{F})$ over all facts and $\overset{*}{F}_b(\mathbb{F} - \{f\})$, excluding the fact f. The consistent reduction in pass@1 (denoted as Δ) emphasises the value of each fact in bug fixing.

To determine the statistical significance of the impact of each individual fact under optimal selection, we calculated pass@1 scores for all bugs, both with and without a particular fact. These scores were then compared. the Wilcoxon signed-rank test shows that the inclusion of each fact has a statistically significant effect on each bug's optimal fact set.

> **Finding 1.** *Under the assumption that we select the most effective fact set for repairing each bug, each of the considered facts demonstrates its utility on our dataset.*

When selecting an optimal fact set for repairing each bug, each of the seven considered facts proves useful on our dataset: including any of these facts in the prompt helps repair at least one bug exclusively and has a statistically significant positive impact on pass@1.

B. Impact of Fact Set Size on Prompt Performance

In this section, we investigate the concept of prompt monotonicity by examining how the incremental addition of facts affects prompt performance. Monotonicity, in this context, refers to a consistent improvement in performance with each additional fact. This implies that more information invariably leads to better outcomes. Conversely, non-monotonicity indicates that there exists a threshold beyond which adding more facts does not enhance, and may even degrade, performance.

Similarly to Section IV-A, we evaluate the non-monotonicity of prompts in two settings: under an optimal fact selection and under a random selection. For each of them, we aggregate pass@1 scores for all bugs over sets containing a varying number of facts. Figure 2 presents the performance of pass@1 for the prompts across different fact set cardinalities, ranging from 0 to 7. The maximum pass@1 corresponds to an optimal fact selection for each bug, and the average pass@1 corresponds to a random fact selection. To ensure a fair comparison across different fact set cardinalities, we sampled 128 elements with replacement for each cardinality. This accounts for the difference in the cardinality of the fact sets we are sampling. Specifically, given two sets of

samples $X_1, \ldots, X_{32}$ and $Y_1, \ldots, Y_{64}$ drawn from the same distribution, $E[max(X_1, \ldots X_{32})] < E[max(Y_1, \ldots, Y_{64})]$ due to the properties of the function max

From the plot, we observe that for both GPT-3.5 and Llama3-70B, the "Max Pass@1" scores (solid lines) generally increase with the number of facts, reaching a peak at 3 facts for GPT-3.5 (solid blue) and 4 facts for Llama3-70B (solid orange) before declining. The "Avg Pass@1" scores (dotted blue line) for GPT-3.5 show improvement with more facts, reaching a plateau between 2 and 5 facts, followed by a decrease. For Llama3-70B, the "Avg Pass@1" scores (dotted orange line) increase up to 4 facts and then decrease. This pattern confirms the non-monotonicity in prompt performance as the number of facts increases.

> **Finding 2.** *Prompt performance is non-monotonic with respect to the number of included facts. While adding facts generally improves performance, there exists a threshold beyond which additional facts hinder performance.*

C. Non-Existence of a Universally Optimal Fact Set

A "universally optimal fact set" in the context of automated program repair is a collection of facts that, when applied, yields the highest effectiveness in terms of bug fixes and pass@1 scores across a wide range of projects. For defining an universally optimal fact set, we first define the quality of a fact set in terms of the following properties:

- **Efficiency:** A fact set is efficient when it outperforms alternative fact sets.
- **Universality:** A fact set is universal when it is efficient up to ϵ-tolerance, as defined in Equation (4).
- **Coverage:** The set of bugs a fact set can resolve.

We define the function $Coverage : 2^{\mathbb{F}} \to 2^B$ where $F \subseteq \mathbb{F}$ is a fact set and B is the set of all bugs in the dataset being considered, such that $Coverage(F)$ returns the set of bugs fixed by the fact set F.

The Coverage Ratio for a given fact set F is defined as:

$$CR(F) = \frac{|Coverage(F)|}{|\bigcup_{\forall F_i \subseteq \mathbb{F}} Coverage(F_i)|} \qquad (6)$$

We prefer sets that maximise universality and coverage ratio.

Table III presents the best-performing fact sets for each project, along with their project-specific and overall pass@1 scores for both GPT-3.5 and Llama3-70B. Notably, no single fact set achieves top performance across all projects. For instance, the fact set 0010101 achieves a pass@1 of 0.21 on GPT-3.5 for the Sanic project, which is below the highest pass@1 score of 0.26. Similarly, in the FastAPI project, the best fact set 0000101 attains a project-specific pass@1 of 0.19 on GPT-3.5, slightly exceeding its overall score of 0.18, yet still lower than the highest score of 0.26 on the same model. A similar variability is observed with Llama3-70B, where no common fact set appears across different repositories, highlighting the unique nature of each project's optimal fact

Project	GPT-3.5			Llama3-70B		
	Best Fact Set	Project	All	Best Fact Set	Project	All
luigi	0011111	0.53	0.26	1010111	0.42	0.16
black	0010111	0.38	0.22	1101011	0.35	0.17
fastapi	0000101	0.19	0.18	0011001	0.11	0.11
httpie	0001111	0.50	0.25	0001000	0.50	0.12
pandas	0011001	0.25	0.25	0101011	0.13	0.17
tornado	0011000	0.37	0.20	1001001	0.22	0.13
ansible	0010100	0.10	0.12	0011011	0.15	0.14
matplotlib	0001001	0.36	0.25	0101010	0.26	0.16
cookiecutter	0001101	0.93	0.20	0001010	0.67	0.14
tqdm	0001111	0.00	0.24	1011001	0.00	0.13
youtube-dl	0011001	0.08	0.25	0001110	0.06	0.14
keras	0101111	0.32	0.23	1011110	0.17	0.15
scrapy	0011111	0.45	0.26	1101011	0.36	0.15
sanic	0010101	0.64	0.21	0110011	0.51	0.15
thefuck	0001001	0.16	0.21	0001001	0.29	0.12

TABLE III: Comparison of project-specific best fact sets and their project's average Pass@1 scores and the total average Pass@1 scores across all projects in BGP157PLY1 . The fact sets are represented using their bitvector encodings. The table highlights that different repositories have different best fact sets, indicating the importance of tailored fact selection for effective bug fixing.

Fact Aggregations	#fixed
Best Fact Set in BGP157PLY1	77
Best Fact Set in BGP157PLY2	84
All Facts	72
Bugs Fixed by the Top 5 Fact Subsets	99
Bugs Fixed by Any Fact Subset	119

TABLE IV: The number of bugs plausibly fixed by fact subset in BGP157PLY2 using GPT-3.5.

set. Additionally, the highest occurrence count for any fact set is just 2, with five fact sets appearing twice. This distribution emphasizes the inconsistency in fact set effectiveness across projects and suggests that a universally optimal fact set is unlikely, given the diverse nature of repositories and bugs.

Table IV assesses the effectiveness of various approaches of fact selection in producing plausible fixes on GPT-3.5. The analysis underscores the Best Fact Set, identified as 1111001, which was selected for its highest coverage in terms of the number of bugs it could fix according to the training data, compared against broader approaches such as the Top 5 Union and Total Union. The Top 5 Union, which aggregates the bugs fixed by the top five best performing fact sets, generates fixes that pass tests for 99 bugs, while the Total Union, encompassing bugs fixed by all fact subsets, resolves 119 bugs. These unions significantly surpass the Best Fact Set in bug resolution capability, fixing many additional bugs (22 and 42 respectively). This shows that the highest coverage ratio of the fact set is $CR(1111001) = 0.65$ that it fixes 65% of the bugs while missing 35% of the bugs fixable by other sets.

These results highlight that the Best Fact Set does not have a high coverage ratio, especially compared to the theoretical maximum of an optimal fact selection which has a coverage ratio of 1. The coverage ratio's delineation as monotonic — in that the fact set with the highest number of bugs fixed is deemed the best — indicates that within this dataset, no fact set achieves a high coverage ratio. Table IV further shows the limitations inherent in static fact selection strategies, as demonstrated by the performance of the Best Fact Sets *w.r.t.*

BGP157PLY2. These sets fix 84 bugs, and set the upper limit for universal fact selection in BGP157PLY2. These sets were not ranked high in the training data, as they were positioned at ranks 16 and 34, respectively out of 128 candidates.

The UpSet diagram [35] in Figure 4 shows the overlap among the top 5 fact sets from BGP157PLY1, as well as a fact-free baseline, encoded as 0000000. The diagram reveals the number of bugs addressed by various intersections of these fact sets, with the largest subset intersection resolving 41 bugs. The diagram shows that each fact set individuallys contribute to the resolution of up to 7 bugs. Cumulatively, these 6 fact sets fix a total of 107 bugs, surpassing the efficacy of the single best fact set, which fixes 89 bugs. This UpSet plot helps us answer the question of the existence of a universal fact set. If one existed, we would expect to see a row with dots in most, if not all, columns, signifying that fact set's presence in the majority of intersections. No such pattern exist in this UpSet plot, indicating that no single fact set fixes all bugs. Instead, different fact sets are effective for different bugs.

Finding 3. *The diversity in the best fact sets across different repositories (Table III), and the significant difference in the bugs fixed by the top five fact sets (Figure 4), both point to the absence of a universal fact set in our dataset.*

D. Effect of Fact Order on Performance

The experiments conducted to this point assume a fixed fact order. We now investigate the effect of permuting the facts. Our prompt template constrains the permutations we consider. We add the facts "Buggy class" (1.1) and "Used method signatures" (1.2) to our prompt next to each other in the order that they appear in source code, so we do not separately permute them. Similarly, the "Failing Tests" (1.3) fact immediately precedes the "Error Information" (2.1) fact it generates in the template. Thus, we only permute 5 facts.

This experiment was conducted on BGP32, comprising 32 bugs. For each of the 120 permutations, we created prompts for each bug, resulting in a total of 120×32 prompts. We computed the pass@1 for each prompt, and then averaged these scores across the 32 bugs for each permutation. The resulting violet histogram in Figure 5 represents the mean pass@1 performance across these permutations.

We conducted a similar analysis for different fact subsets, also shown in Figure 5. With 7 facts, there are 128×32 prompts. For each subset, we calculated the mean pass@1 across the 32 bugs. The yellow histogram represents the distribution of the mean performance over these fact subsets. Comparing the two histograms reveals that the variability due to fact order (violet) is relatively narrow, ranging from 0.24 to 0.31. In contrast, the impact of different fact subsets (yellow) is significantly broader, ranging from 0.07 to 0.34. This demonstrates that the selection of which facts to include (fact subsets) has a much greater influence on prompt performance than the specific order in which these facts are presented.

Fig. 4: This upset diagram compares the 5 fact sets that fix the most bugs in BGP157PLY1 with using no facts (encoded as 0000000) including the buggy function and chain-of-thought instructions. The fact sets shown on the left collectively fix 107 bugs, 18 more than the single best fact set, which fixes 89 bugs. The diagram is constructed by generating prompts from each fact set, sending them to GPT-3.5 for $n = 15$ responses, and identifying the set of bugs with at least one passing response.

Fig. 5: Distribution of mean pass@k evaluated on BGP32. The violet histogram reports the mean performance under permutation; the yellow histogram reports mean by subset. The wider range in the yellow distribution suggests that fact subsets have a stronger impact on performance than fact order.

Finding 4. *Selecting facts has more impact on performance than ordering them.*

V. SELECTING FACTS WITH MANIPLE

Universal fact selection, as demonstrated in Section IV-C, does not achieve consistent performance across all the bugs in our dataset. Thus, to automate creating bug-tailored prompts, we introduce MANIPLE, a random forest trained to select relevant facts for inclusion in the prompts.

We focus on the task of predicting the success or failure of test executions based on vectors representing features extracted from both the prompt and the code.

Our training dataset $\mathcal{D}$ is constructed from the BGP157PLY1. It consists of pairs (j, y), where:

- $j = (b, F) \in \mathbb{J}$ represents a job, which is a tuple consisting of a bug b along with and a fact set $F \subseteq \mathbb{F}$
- $y \in [0, 1]$ represents the probability of successfully fixing the bug b in a single trial, given the fact set $F \subseteq \mathbb{F}$. This probability is computed using pass@1.

$\mathcal{D}$ consists of 157 bugs, each with 128 fact combinations, totaling $20,096$ samples. However, since not all facts are available for every bug, missing facts are labeled as "None". This reduces the number of unique prompts to 9496.

We manually craft a set of features $\mathbf{f}(j)$ based on domain knowledge and the characteristics of the facts. The feature function $\mathbf{f} : \mathbb{J} \rightarrow \mathbb{R}^m$ maps the input job $j \in \mathbb{J}$ to an m-dimensional feature space.

The goal is to train a machine learning model $\mathcal{M}$ capable of using the feature vector $\mathbf{f}(j)$ to accurately predict the probability of the success (pass@k) for each fact set. Using $\mathcal{M}$, we can select the optimal fact set by choosing the one that yields the highest predicted probability of success according to the model's output.

A. Feature Selection

To train a machine learning model to meet these objectives, we define the feature vector, $\mathbf{f}(\mathbf{x}) = [\mathbf{b}, rep\_id, \ell, c]^T$, where

- Bitvector ($\mathbf{b}$): encodes the fact set F, where $\mathbf{b} \in \{0, 1\}^n$.
- Repository ID (rep_id): uniquely identifies the source repository of the bug b.
- Prompt Length (ℓ): the length of the prompt in either characters or tokens, where $\ell \in \mathbb{N}^0$. Cross-validation determines whether characters or tokens are chosen.
- Cyclomatic Complexity (c): a measure of code complexity that quantifies the number of linearly independent paths through a program's source code.

This choice of features was guided by the investigation conducted on BGP157PLY1. Specifically, the including Repository ID (rep_id) is supported by the findings in Table III, which illustrate significant variability in the optimal bitvectors for fact selection across different projects. This variability underscores the influence of the repository context on successful repair. Additionally, Prompt Length (ℓ) was identified as a crucial factor, displaying a Spearman correlation of -0.18 with the pass@1 for repair success, accompanied by a highly significant p-value of $p < 10^{-129}$. This correlation holds for both token and character lengths of prompts, indicating that an increase in prompt length is associated with a decrease in LLM performance — a conclusion that is further supported by the non-monotonicity of adding facts to prompts, as noted in Figure 2, as more facts increase prompt length.

Cyclomatic Complexity emerged as another pivotal feature, showing a negative Spearman correlation of -0.1 with the pass@1, p-value of 10^{-42}. Unlike prompt length, Cyclomatic complexity does not depend on the fact set chosen (recall that the buggy code itself is necessarily always included) but remains instrumental in predicting the pass@1 for a bug, mainly for scenarios where no prompt generates a successful fix.

B. MANIPLE: A Random Forest for Fact Set Selection

Leveraging these observations, we present MANIPLE, a random forest model for the fact selection task and evaluate it in both regression and classification settings. As a regressor, the model directly predicts the pass@1 for each job. In the classification task, we categorise the scale (i.e., [0, 1]) based on the number of fact sets considered by the model. Our analysis reveals that classification performs better. This finding can be attributed to the noise and significant variance present in our data, as detailed in Appendix A of the full version of this paper on arXiv [28]. The classification method proved more robust to variance in pass@1 compared to regression. Additionally, ordering and ranking the fact sets did not yield comparable performance. This is likely due to the variance in ranks, which directly stems from the variance in pass@1.

Additionally, we trained MANIPLE only on the top five highest-performing fact sets. These sets were identified using bootstrap aggregation, where we evaluated the performance of each fact set across multiple bootstrap samples drawn from our dataset. By aggregating the outcomes, we identified the best performing fact sets. We optimised the model's hyperparameters through a comprehensive grid search, evaluating the performance of each combination of parameters. To ensure the model's generalisability and to prevent overfitting, we employed k-fold cross-validation with $k = 5$.

VI. COMPARING MANIPLE WITH SOTA LLM-BASED APR

We compared MANIPLE with existing zero-shot non-conversational LLM-based APR methods that incorporate various types of information into prompts. Our baselines include approaches whose prompts include the following facts:

- Buggy Function only, denoted as T_0
- Buggy Function, Buggy Class and Used Method Signatures, denoted as T_1, an approach similar to the technique by Chen *et al.* [6].
- Buggy Function combined with GitHub Issue, denoted as T_2, an approach similar to the technique by Fakhoury [3].
- Buggy Function alongside Error Information, denoted as T_3, an approach similar to the technique by Keller *et al.* [7].

For a fair comparison, we supply the same prompts, built from these facts, to MANIPLE and all the baselines. Unsurprisingly, these prompts differ structurally from the prompts on which the baselines were run. For example, our prompt incorporates our chain-of-thought instructions defined in Appendix C of the full version of this paper on arXiv [28]. Our focus here is on comparing MANIPLE's performance to

Tool	#fixed	Correct	% Correct
T_0	44	7	16%
T_1	37	5	14%
T_2	63	18	29%
T_3	66	31	47%
MANIPLE	88	37	42%

TABLE V: Comparison of tool performance on the BGP157PLY2 test set, focusing on bugs fixed with GPT-3.5. **#fixed** represents the number of bugs with test-passing patches, and **Correct** indicates bugs fixed with patches identical to the developer's. We observe that MANIPLE outperforms all the fact set combinations used. This shows that bug-tailored fact selection can improve the repair success.

the baselines' *w.r.t.* facts, not their performance given approach-specific optimal prompts, whose construction would require close cooperation with the authors of each baseline.

We evaluated tool efficacy according to two criteria, based on $n = 15$ responses (Section III): (1) number of plausible fixes (*i.e.* eq. (2)), and (2) the number of correct fixes, determined by the first plausible patch generated by the LLM and subsequently subjected to manual evaluation.

For assessing patch correctness, we employ an interrater agreement scale, where "3" indicates patches syntactically equivalent to the developer's patch, allowing for minor refactoring or restructuring, "2" — patches that achieve the intended outcome through an alternate method, "1" — diverging from the developer's patch, rendering correctness indeterminable, and "0" — incorrect patches (irrelevant, incomplete, introducing regressions). Each patch undergoes evaluation by two independent raters. In cases of scoring discrepancies, the raters discuss them. If the discrepancy is not resolved, the more conservative (lower) score is recorded, ensuring a rigorous standard of correctness. Following this scoring system, patches with the scores of 2 and 3 are labelled as "correct", and with the scores of 1 and 0 are labelled as "incorrect."

The analysis in Table V underscores the potential of bug-tailored prompt selection for boosting automated program repair efficacy. Previous LLM-based zero-shot non-conversational approaches, represented by T_1, T_2 and T_3, that always use the same set of facts achieve a maximum of 66 plausibly fixed patches on BGP157PLY2. In contrast, MANIPLE, which leverages bug-tailored fact selection, identifies a significantly higher number of plausible patches (88). Furthermore, MANIPLE boasts 6 additional correct fixes compared to the best of these approaches. These findings suggest that tailoring the fact set to the specific context of each bug has the potential to improve the upper bound for repair success.

Surprisingly, T_1, which utilises function code alongside scope information (including class information and invoked functions), fixes fewer bugs compared to using function code alone. However, this does not imply that scope and class information are useless. T_1 still identifies 5 plausible patches, 2 of which are correct, that would not be found using just the function code as the only fact.

VII. THREATS TO VALIDITY

In assessing LLMs, we recognise the challenge of potential data leakage, as some training datasets are not publicly avail-

able. This limits our ability to know exactly what information the model has encountered. However, our primary focus is on the impact of different facts on repair performance, rather than the performance itself. For instance, if a prompt with an "angelic forest" leads to a bug fix while a prompt without it does not, this highlights the significant role of "angelic forest" in enhancing bug-fixing performance. This remains true regardless of the bug's presence in the training data. Thus, our findings are robust even with possible data leakage, and using the same model across our baselines partially mitigates this issue.

We extracted our fact set from the APR literature. We acknowledge that it is provisional and likely to change as research into LLM-assisted APR continues.

Our dataset is a subset of BugsInPy, a benchmark published at FSE'20 [10]. BugsInPy is a curated dataset built to best practice from GitHub repos with more than 10k stars that have at least one test case in the fixed commit that distinguishes the buggy version from its fix. Its representativeness has not been challenged and, from first principles, we can think of no reason that our filtering (Section III-A) would introduce systematic bias.

VIII. RELATED WORK

Traditional APR uses search, *e.g.* GenProg [36], or program synthesis such as SemFix [37], Angelix [9], SE-ESOC [38], and Trident [39]. This study includes Angelic forests [9] in its fact set, demonstrating their utility in LLM-based APR.

CoCoNut [21] utilizes surrounding contextual information to train an ensemble of neural machine translation models. Rete [40] employs Conditional Def-Use chains as context for CodeBert [41]. CapGen [42] utilises AST node information to estimate the likelihood of patches. DLFix [43] treats the program repair task as a code transformation task, learning to transform by additionally incorporating the surrounding context of the bug. The context used in this work includes the class and scope information of the buggy program, which is broader than the contexts used above. FitRepair [2] constructs prompts using identifier extracted from lines that look similar to the buggy line. Although, our work does not directly provide identifiers statically, as python is dynamic, we provide dynamic values of the variables during the test run.

LLM-based Program Repair: APR leveraging LLMs is making strides. InferFix [44] uses few-shot prompting to repair issues from Infer. ChatRepair [4] uses interactive prompting constructed from failing test names and their failing assertions. Our approach focuses on the zero-shot, non-conversational setting, but can be potentially integrated with InferFix and ChatRepair. Various approaches focused on program engineering for APR, *e.g.* incorporating bug-related information within the prompts [44], [45], [46], as the quality of fixes could be enhanced by integrating contextual information, such as the bug's local context [5] and details about relevant identifiers [2], into the prompt. Xia *et al.* [2] utilize relevant identifiers to augment the fix rate of prompts. Keller *et al.* [7] reveals that, for debugging tasks, focusing on the specific line indicated by a stack trace is more effective than providing the entire trace. Similarly, Fakhoury *et al.* [3] investigates how combining issue descriptions with bug report titles and descriptions enhances program repair efforts. Following recent research [1], we refer to such pieces of information as *facts*. Our work uses individual facts from previous work to formulate and motivate the fact selection problem.

Program Repair for Python: QuixBugs [47] is a benchmark consisting of small programs in Java, and Python. They are not reflective of real software projects. Bugswarm [48], was constructed by automatically mining failing CI builds, and thus contains issues outside of the scope of our study, such as configuration issues. BugsInPy [10] manually curates 501 bugs from 17 popular Python Projects. We selected 314 bugs from this benchmark that require modifications within a single function, and which we managed to reproduce. Rete [40], a program repair tool that leverages contextual information, was evaluated on both Python and C, using BugsInPy as its Python benchmark. We did not compare MANIPLE with Rete, because it relied on the line-granular perfect fault localisation (PFL), while this study uses function granular PFL. PyTER [49] is a program repair technique that focuses on Python TypeErrors; it was evaluated on a custom benchmark for type errors.

Prompt Engineering: Recent advancements in prompt engineering have significantly influenced the effectiveness of models like ChatGPT. Notably, the tree-of-thoughts approach [50] and the zero-shot-CoT approach [51] have emerged as pivotal strategies. Frameworks like ReACT [52] use LLMs to generate interleave reasoning and task-specific actions. Self Consistency [53] traverses multiple diverse reasoning paths through few shot CoT and uses the generations to select the most consistent answer. Automatic Prompt Engineer [54] frames prompt constructs as a natural language synthesis task. It is orthogonal to our approach, as our task is to select ideal facts while Automatic Prompt Engineer seeks to refine the prompt into which the selected facts can be plugged. Repository Level Prompt Generation (RLPG) [55] is a general framework for retrieving relevant repository context and constructing prompts, instantiated for code completion. RLPG uses a classifier to choose the best one. In contrast, our fact selection problem aims to find an optimal combination of facts to repair a given bug. In addition to repository context, our work incorporates dynamic and information external to the code repository.

IX. CONCLUSION

In this paper, we explore the construction of effective prompts for LLM-based APR, leveraging facts extracted from the buggy program and external sources. Through a systematic investigation, we incorporate seven bug-relevant facts into the prompts, notably including angelic values, a factor previously not considered in this domain. Furthermore, we define the fact selection problem and demonstrate that a universally optimal set of facts for addressing various bugs does not exist. Building on this insight, we devise a bug-tailored fact selection strategy enhancing the effectiveness of APR.

REFERENCES

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

[2] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," *arXiv preprint arXiv:2303.10494*, 2023.

[3] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," *arXiv preprint arXiv:2304.03816*, 2023.

[4] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[5] J. A. Prenner and R. Robbes, "Out of context: How important is local context in neural program repair?" in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, 2024.

[6] Y. Chen, J. Wu, X. Ling, C. Li, Z. Rui, T. Luo, and Y. Wu, "When large language models confront repository-level automatic program repair: How well they done?" *arXiv preprint arXiv:2403.00448*, 2024.

[7] J. Keller and J. Nowakowski, "Ai-powered patching: the future of automated vulnerability fixes," Tech. Rep., 2024.

[8] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 121–130.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[10] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.

[11] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *arXiv preprint arXiv:2307.03172*, 2023.

[12] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[13] "The pandas:128 bug from bugsinpy," https://github.com/pandas-dev/pandas/commit/112e6b8d054f9adc1303138533ed6506975f94db, 2023, accessed: 2024-03-21.

[14] J. Schulman, B. Zoph, C. Kim, J. Hilton, J. Menick, J. Weng, a. Felipe Juan 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, N. Ryder, A. Paino, Q. Yuan, C. Winter, B. Wang, M. Bavarian, I. Babuschkin, S. Sidor, I. Kanitscheider, M. Pavlov, M. Plappert, N. Tezak, H. Jun, W. Zhuk, V. Pong, L. Kaiser, J. Tworek, A. Carr, L. Weng, S. Agarwal, K. Cobbe, V. Kosaraju, A. Power, S. Polu, J. Han, R. Puri, S. Jain, B. Chess, C. Gibson, O. Boiko, E. Parparita, A. Tootoonchian, K. Kosic, and C. Hesse, "Introducing chatgpt," *OpenAI blog*, Nov 2022. [Online]. Available: https://openai.com/blog/chatgpt

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[16] L. Weng, "Prompt engineering," *lilianweng.github.io*, Mar 2023. [Online]. Available: https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/

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

[18] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" *arXiv preprint arXiv:2101.06804*, 2021.

[19] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[20] 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)*. IEEE, 2007, pp. 89–98.

[21] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

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

[23] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[24] "Pandas, python data analysis library," https://pandas.pydata.org/, 2023, accessed: 2024-03-21.

[25] "Matplotlib: Visualization with python," https://matplotlib.org/, 2023, accessed: 2024-03-21.

[26] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.

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

[28] N. Parasaram, H. Yan, B. Yang, Z. Flahy, A. Qudsi, D. Ziaber, E. Barr, and S. Mechtaev, "The fact selection problem in llm-based program repair," *arXiv preprint arXiv:2404.05520*, 2024.

[29] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[30] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[31] "Openai api," https://openai.com/blog/openai-api, 2024, accessed: 2024-03-21.

[32] "Deepinfra api," https://deepinfra.com/, 2024, accessed: 2024-07-21.

[33] E. Winter, "The shapley value," *Handbook of game theory with economic applications*, vol. 3, pp. 2025–2054, 2002.

[34] "Relative change," https://en.wikipedia.org/wiki/Relative_change, 2023, accessed: 2024-03-21.

[35] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "Upset: visualization of intersecting sets," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 1983–1992, 2014.

[36] 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, no. 1, pp. 54–72, 2011.

[37] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[38] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, "Symbolic execution with existential second-order constraints," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 389–399.

[39] N. Parasaram, E. T. Barr, and S. Mechtaev, "Trident: Controlling side effects in automated program repair," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4717–4732, 2021.

[40] ——, "Rete: Learning namespace representation for program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1264–1276.

[41] 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," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[42] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1–11.

[43] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[44] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," *arXiv preprint arXiv:2303.07263*, 2023.

[45] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022.

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

[47] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.

[48] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.

[49] W. Oh and H. Oh, "Pyter: effective program repair for python type errors," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 922–934.

[50] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.

[51] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[52] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

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

[54] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," *arXiv preprint arXiv:2211.01910*, 2022.

[55] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.