Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction

Sungmin Kang*, Juyeon Yoon*, Nargiz Askarbekkyzy, and Shin Yoo

Abstract—Bug reproduction is a critical developer activity that is also challenging to automate, as bug reports are often in natural language and thus can be difficult to transform to test cases consistently. As a result, existing techniques mostly focused on crash bugs, which are easier to automatically detect and verify. In this work, we overcome this limitation by using large language models (LLMs), which have been demonstrated to be adept at natural language processing and code generation. By prompting LLMs to generate bug-reproducing tests, and via a post-processing pipeline to automatically identify promising generated tests, our proposed technique LIBRO could successfully reproduce about one-third of all bugs in the widely used Defects4J benchmark. Furthermore, our extensive evaluation on 15 LLMs, including 11 open-source LLMs, suggests that open-source LLMs also demonstrate substantial potential, with the StarCoder LLM achieving 70% of the reproduction performance of the closed-source OpenAI LLM code-davinci-002 on the large Defects4J benchmark, and 90% of performance on a held-out bug dataset likely not part of any LLM's training data. In addition, our experiments on LLMs of different sizes show that bug reproduction using LIBRO improves as LLM size increases, providing information as to which LLMs can be used with the LIBRO pipeline.

Index Terms—test generation, natural language processing, software engineering

1 Introduction

Code will often contain bugs; consequently, bug tracking or bug reporting software is in popular use among developers [1]. When dealing with bug reports submitted via the tracker, one important activity is bug reproduction [2], in which a developer tries to reproduce the reported buggy behavior by encapsulating it in a test case. Such bug-reproducing tests have substantial value both for developers and automated debugging techniques: prior work has shown that developers make extensive use of bug reproducing tests (BRTs) when debugging [3], while many automated debugging techniques also assume the existence of BRTs as a prerequisite for operation [4].

Due to the importance of this task, existing work has suggested automated bug reproduction techniques, which take a bug report (usually in natural language) as input, and return a bug-reproducing test as output. However, all such techniques target crash bugs, in which the test oracle is clear [5]; in contrast, the research on reproducing functional bugs has been sparse. This is perhaps due to the difficulty of the level of natural language processing and code synthesis that is necessary to successfully reproduce a bug: an automated technique would need to reliably translate the functional specification within the bug report (which may be in natural language) to an executable test case.

In this work, we first argue that large language models (LLMs), which have demonstrated substantial performance in natural language processing [6] and code synthesis [7], may provide a useful solution to this problem. To this end, we propose LIBRO, a pipeline that uses LLMs to first generate a number of BRT candidates via constructing an appropriate prompt with a given bug report. All generated tests are subsequently evaluated by the postprocessing pipeline of LIBRO. It will automatically filter low-quality tests that do not fail, and sort the remaining tests based on test execution features to minimize developer inspection cost of automatically generated tests; this is critical for improving developer experience and adoption [8].

Our experiments reveal that when using the code-davinci-002 model from OpenAI, LIBRO can reproduce one-third of all the bugs in the widely used Defects4J benchmark [9], substantially outperforming bug reproduction baselines that we compared against. Furthermore, our post-processing pipeline could successfully identify cases where LIBRO was more likely to yield accurate bug reproduction results, allowing users to control the precision of LIBRO up to 90% and to place an actual BRT as the first-place suggestion in 60% of all bugs that could be reproduced.

This work is an extension of our previous publication [10]. Relative to our previous publication, this extension primarily contributes a newly performed extensive study comparing the capabilities of 15 LLMs in total, including 11 open-source LLMs; to the best of our knowledge, we are the first to perform such a large-scale comparison of open source LLMs on testing. Our LLM comparison also allows us to perform multiple analyses that were impossible to do in our previous work: (1) we plot the trade-off between model GPU memory consumption and performance, which can help practitioners decide which LLM to use given their computational resources; (2) evaluate the influence of model

This work was supported by the National Research Foundation of Korea (NRF) Grant (RS-2023-00208998), Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (2021R1A5A1021944), and the Institute for Information & Communications Technology Promotion grant funded by the Korean government MSIT (IITP2022-0-00995).

- *S. Kang, J. Yoon, and S. Yoo are with the Korea Advanced Institute of Science and Technology (KAIST). N. Askarbekkyzy contributed while at KAIST.*
- *S. Kang and J. Yoon contributed equally to this publication.*

size on performance when the training technique is controlled; (3) discuss how the output of ChatGPT has changed over time, and (4) explore whether self-consistency [11], the basis for our post-processing pipeline, works for other LLMs as well. The LLM comparisons made in this paper required more than eight months of GPU time and seven months of CPU time to be fully executed. The specifics of our findings are detailed below:

- Through experiments with multiple LLMs, we demonstrate that LIBRO is a general technique that is not only effective when using a particular LLM.
- Through our comparison of 15 LLMs, we find that among the closed-source LLMs which we could experiment with, the `code-davinci-002` model showed the best performance, while among the open-source LLMs, the recent StarCoder model [12] showed the best performance, reproducing about 70% of the bugs that `code-davinci-002` could reproduce. Furthermore, we demonstrate that open-source LLMs can make bug-reproducing tests outside of their training data.
- By comparing LLMs from the same family trained with different data or parameters, we find that fine-tuning code LLMs on natural language can hurt performance, and that larger models tend to show better performance, suggesting guidelines on what code LLMs to use when operating LIBRO.
- We experiment with the temperature parameter of LLMs, which controls the randomness of the LLM output, and find that a value of 0.6 works best.
- We make our experimental data and analysis scripts publicly available: https://github.com/coinse/libro-journal-artifact.

The remainder of the paper is organized as follows. We motivate our research in Section 2. Based on this, we describe our approach in Section 3. Evaluation settings and research questions are in Section 4 and Section 5, respectively. Results are presented in Section 6, Section 7 provides in-depth discussion of our technique, Section 8 gives an overview of the relevant literature, and Section 9 concludes.

2 MOTIVATION

2.1 Bug Reproduction

As described in the previous section, generating bug-reproducing tests from bug reports is important for both developers and automated techniques. First, Koyuncu et al. [4] report that Spectrum-Based Fault Localization (SBFL) techniques cannot locate the bug at the time of being reported in 95% of the cases they analyzed, due to the lack of a bug-reproducing test when the report was first filed. Other studies show that developers use bug-reproducing tests extensively when doing debugging themselves: for example, Beller et al. [3] note that 80% of developers would use bug-reproducing tests to verify fixes. Automatic bug reproduction is also important as the report-to-test problem is a perhaps underappreciated yet nonetheless important and recurring part of testing. Kochhar et al. [13], for example, explicitly ask hundreds of developers on whether they agree to the statement “during maintenance, when a bug is fixed, it is good to add a test case that covers it”, and find a strong average agreement of 4.4 on a Likert scale of 5.

To further verify that developers regularly deal with the report-to-test problem, we analyze the number of test additions that can be attributed to a bug report, by mining hundreds of open-source Java repositories. We start with the `Java-med` dataset from Alon et al. [14], which consists of 1000 top-starred Java projects from GitHub. From the list of commits in each repository, we check (i) whether the commit adds a test, and (ii) whether the commit is linked to an issue. To determine whether a commit adds a test, we check that its diff adds the `@Test` decorator along with a test body. In addition, we link a commit to a bug report (or an *issue* in GitHub) if (i) the commit message mentions "(fixes/resolves/closes) #NUM", or (ii) the commit message mentions a pull request, which in turn mentions an issue. We compare the number of tests added by such report-related commits to the size of the test suite at the time of gathering (August 2022) to estimate the prevalence of such tests. As different repositories have different issue-handling practices, we filter out repositories that have no issue-related commits that add tests, as this indicates a different bug handling practice (e.g. `google/guava`). Accordingly, we analyze 300 repositories, as shown in Table 1.

TABLE 1: Analyzed repository characteristics

Repository Characteristic	# Repositories
Could be cloned	970
Had a JUnit test (`@Test` is found in repository)	550
Had issue-referencing commit that added test	300

Overall, in these 300 repositories that we inspected, the number of tests that were added by issue-referencing commits was on median 28.4% of the overall test suite size. While one must note that due to code evolution, this may not mean that 28.4% of all current tests were added by bug reports, overall these results suggest a substantial portion of bugs are being added via bug reports, supporting our claim that test addition after bug reproduction is a common task that developers face. In turn, automating the report-to-test activity could provide substantial benefit to developers, and would help them within their existing workflow.

However, the problem has been difficult for researchers to tackle until recently, due to the difficulties of natural language processing and corresponding code synthesis. For example, the bug report in Table 2 does not explicitly specify any code, which would make this report difficult to automatically process for traditional techniques, even though a user fluent in English and Java would be capable of deducing that when both arguments are NaN, the ‘equals’ methods in ‘MathUtils’ should return `false`. As a result, most existing work focused on reproducing crashes [15], [16].

Large Language Models (LLMs), which have been pre-trained on large corpora of natural language and programming language data, may provide a potential solution to tame this difficulty. LLMs show a surprising level of performance on both natural language processing [6] and software engineering [10], [17] tasks. Their capability, and the recent introduction of many different open-source LLMs as described in the next subsection, thus poses the question of whether this emerging technology could be used to alleviate developer effort in writing tests from bug reports.

2.2 Open-source Large Language Models

While Brown et al. [6] first demonstrated the substantial potential of LLMs, the LLMs from OpenAI have not since been made public - that is, they have only been made available via an API, without any sharing of source code or neural network weights. This makes consistent research using LLMs difficult, as LLMs accessed through an API can see significant behavior change in a short period of time, as Chen et al. [18] identify. Indeed, earlier this year, the API to the `code-davinci-002` model was discontinued [19], making previous papers that utilized the model (including our previous work [10]) difficult to reproduce. In reaction to the secrecy and potential centralization threatening the open spirit of research, multiple open-source LLMs have been suggested [20], [12]. Unlike the widely-used OpenAI models, which require a per-usage fee for their API use and which are not available in all countries, the LLMs that we use are generally free for use in research in all countries, and can be operated without cost provided that one has the necessary computational resources. As a result, while much research is using OpenAI models such as ChatGPT [19], we believe that it would benefit the software engineering community to evaluate open-source LLMs under the same task as closed-source LLMs and compare their performance, to showcase their viability and promote their widespread use in research.

3 Approach

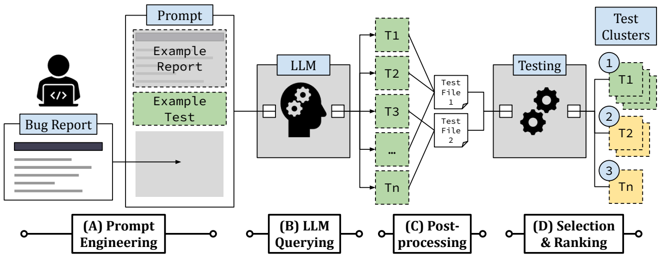


Fig. 1: Overview of Libro

Figure 1 presents a schematic of our approach, Libro. In step (A), a bug report is used to construct a 'prompt' which conditions an LLM to generate a bug-reproducing test corresponding to the content of the bug report. This prompt is used in step (B), where an LLM generates multiple candidate bug-reproducing tests based on the prompt. However, showing all such results to a developer would be overwhelming, so tests are executed. First, in step (C), generated tests are injected into the existing test suite so that they can be executed; then, in step (D), execution results are used to filter out and rank generated tests so that developers only need to inspect the most promising generated tests. In the remainder of this section, we describe each step in greater detail using the example provided in Table 2.

3.1 Prompt Engineering

LLMs are, at the core, large autocomplete neural networks: prior work has found that different ways of 'asking' the LLM to solve a problem will lead to significantly varying levels of performance [21]. Finding the best query to accomplish the given task is known as *prompt engineering* [22].

To make an LLM generate a test method from a given bug report, we construct a Markdown document from the report, which is used as the prompt. For example, consider the example in Listing 1, which is a Markdown document constructed from the bug report shown in Table 2. Libro adds a few distinctive parts to the Markdown document: the command "Provide a self-contained example that reproduces this issue", the start of a block of code in Markdown, (i.e., ```), and finally the partial code snippet `public void test` which induces the LLM to write a test method.

TABLE 2: Example bug report (Defects4J Math-63).

Issue No.	MATH-370[1]
Title	NaN in "equals" methods
Description	In "MathUtils", some "equals" methods will return true if both argument are NaN. Unless I'm mistaken, this contradicts the IEEE standard. If nobody objects, I'm going to make the changes.

Listing 1: Example prompt without examples.

~~~
# NaN in "equals" methods
## Description
In "MathUtils", some "equals" methods will return true if both argument are NaN.
Unless I'm mistaken, this contradicts the IEEE standard.
If nobody objects, I'm going to make the changes.

## Reproduction
>Provide a self-contained example that reproduces this issue.
```
public void test
~~~

We evaluate a range of variations of this basic prompt. Brown et al. [6] report that LLMs benefit from question-answer examples provided in the prompt. In our case, this means providing examples of bug reports (questions) and the corresponding bug reproducing tests (answers). With this in mind, we experiment with a varying number of examples, to see whether adding more examples, and whether having examples from within the same project or from other projects, significantly influences performance.

As there is no restriction to the prompt format, we also experiment with providing stack traces for crash bugs (to simulate situations where a stack trace was provided), or providing constructors of the faulty class (to simulate situations where the location of the bug is known).

Our specific template format makes it highly unlikely that prompts we generate exist verbatim within the LLM training data. Further, most reports in practice are only connected to the bug-revealing test via a chain of references. As such, our format partly mitigates data leakage concerns, among other steps taken to limit this threat described later in the manuscript. Finally, when using chat-optimized models such as ChatGPT in our experiments, we add examples

[1] https://issues.apache.org/jira/browse/MATH-370

within the prompt, although in our experiments we needed to update the prompt format as the model was updated, as discussed in Section 6.4.4.

3.2 Querying an LLM

Using the generated prompt, LIBRO queries the LLM to predict the tokens that would follow the prompt. Due to the nature of the prompt, it is likely to generate a test method, especially as our prompt ends with the sequence `public void test`. We ensure that the result only spans the test method by accepting tokens until the first occurrence of the string ```, which indicates the end of the code block in Markdown.

It is known that LLMs yield inferior results when performing completely greedy decoding (i.e., decoding strictly based on the most likely next token) [6]: they perform better when they are doing weighted random sampling, a behavior modulated by the *temperature* parameter. Following prior work, we set our temperature to 0.7 [6], which allows the LLM to generate multiple distinct tests based on the exact same prompt, although we evaluate the effect of temperature in Section 6.4.5 and Section 6.5.2. We take the approach of generating multiple candidate reproducing tests, then using their characteristics to identify how likely it is that the bug is actually reproduced.

An example output from the LLM given the prompt in Listing 1 is shown in Listing 2: at this point, the outputs from the LLM typically cannot be compiled on their own, and need other constructs such as import statements. We next present how LIBRO integrates a generated test into the existing test suite to make it executable.

Listing 2: Example LLM result from the bug report described in Table 2.

```java
public void testEquals() {
    assertFalse(MathUtils.equals(Double.NaN, Double.NaN));
    assertFalse(MathUtils.equals(Float.NaN, Float.NaN));
}
```

3.3 Test Postprocessing

We first describe how LIBRO injects a test method into an existing suite, then how LIBRO resolves the remaining unmet dependencies.

3.3.1 Injecting a test into a suitable test class

If a developer finds a test method in a bug report, they will likely insert it into a test class which will provide the required context for the test method (such as the required dependencies). For example, for the bug in our running example, the developers added a reproducing test to the `MathUtilsTest` class, where most of the required dependencies are already imported, including the focal class, `MathUtils`. Thus, it is natural to also inject LLM-generated tests into existing test classes, as this matches developer workflow, while resolving a significant number of initially unmet dependencies.

Listing 3: Target test class to which the test in Listing 2 is injected.

```java
1  public final class MathUtilsTest extends TestCase {
2      ...
3      public void testArrayEquals() {
4          assertFalse(MathUtils.equals(new double[] { 1d }, null));
5          assertTrue(MathUtils.equals(new double[] {
6              Double.NaN, Double.POSITIVE_INFINITY,
7          ...
```

Algorithm 1: Test Postprocessing

```
   Input: A test method $tm$; Test suite $\mathcal{T}$ of SUT; source code files $\mathcal{S}$ of SUT;
   Output: Updated test suite $\mathcal{T}'$
1  $c_{best}$ ← findBestMatchingClass($tm$, $\mathcal{T}$);
2  $deps$ ← getDependencies($tm$);
3  $needed\_deps$ ← getUnresolved($deps$, $c_{best}$);
4  $new\_imports$ ← set();
5  for $dep$ in $needed\_deps$ do
6      $target$ ← findClassDef($dep$, $\mathcal{S}$);
7      if $target$ is null then
8          $new\_imports$.add(findMostCommonImport($dep$, $\mathcal{S}$, $\mathcal{T}$));
9      else
10         $new\_imports$.add($target$);
11 $\mathcal{T}'$ ← injectTest($tm$, $c_{best}$, $\mathcal{T}$);
12 $\mathcal{T}'$ ← injectDependencies($new\_imports$, $c_{best}$, $\mathcal{T}'$);
```

To find the best test class to inject our test methods into, we find the test class that is *lexically* most similar to the generated test (Algorithm 1, line 1). The intuition is that, if a test method belongs to a test class, the test method likely uses similar methods and classes, and is thus lexically related, to other tests from that test class. Formally, we assign a matching score for each test class based on Equation (1):

$$sim_{c_i} = |T_t \cap T_{c_i}|/|T_t| \tag{1}$$

where T_t and T_{c_i} are the set of tokens in the generated test method and the ith test class, respectively. As an example, Listing 3 shows the key statements of the `MathUtilsTest` class. Here, the test class contains similar method invocations and constants with those used by the LLM-generated test in Listing 2, particularly in lines 4 and 6.

As a sanity check, we inject ground-truth developer-added bug reproducing tests from the Math and Lang projects of the Defects4J benchmark, and check if they execute normally based on Algorithm 1. We find execution proceeds as usual for 89% of the time, suggesting that the algorithm reasonably finds environments in which tests can be executed.

3.3.2 Resolving remaining dependencies

Although many dependency issues are resolved by placing the test in the right class, the test may introduce new constructs that need to be imported. To handle these cases, LIBRO heuristically infers packages to import.

Line 2 to 10 in Algorithm 1 describe the dependency resolving process of LIBRO. First, LIBRO parses the generated test method and identifies variable types and referenced class names/constructors/exceptions. LIBRO then filters "already imported" class names by lexically matching names to existing import statements in the test class (Line 3).

As a result of this process, we may find types that are not resolved within the test class. LIBRO first attempts to find public classes with the identified name of the type; if there is exactly one such file, the classpath to the identified class is derived (Line 7), and an import statement is added

(Line 11). However, either no or multiple matching classes may exist. In both cases, LIBRO looks for `import` statements ending with the target class name within the project (e.g., when searching for `MathUtils`, LIBRO looks for `import .* MathUtils;`). LIBRO selects the most common import statement across all project source code files. Additionally, a few rules ensure assertion statements are properly imported, even when there are no appropriate imports within the project itself.

Our postprocessing pipeline does not guarantee compilation in all cases, but the heuristics used by LIBRO are capable of resolving most of the unhandled dependencies of a raw test method. After going through the postprocessing steps, LIBRO executes the tests to identify candidate bug reproducing tests.

3.4 Selection and Ranking

A test is a Bug Reproducing Test (BRT) if and only if the test fails due to the bug specified in the report. A *necessary* condition for a test generated by LIBRO to be a BRT is that the test compiles and fails in the buggy program: we call such tests FIB (Fail In the Buggy program) tests. However, not all FIB tests are BRTs, making it difficult to tell whether bug reproduction has succeeded or not. This is one factor that separates us from crash reproduction work [15], as crash reproduction techniques can confirm whether the bug has been reproduced by comparing the stack traces at the time of crash. On the other hand, it is imprudent to present all generated FIB tests to developers, as asking developers to iterate over multiple solutions is generally undesirable [23], [24]. As such, LIBRO attempts to decide when to suggest a test and, if so, which test to suggest, using several patterns we observe to be correlated to successful bug reproduction.

Algorithm 2 outlines how LIBRO decides whether to present results and, if so, how to rank the generated tests. In Line 1-10, LIBRO first decides whether to show the developer any results at all (selection). We group the FIB tests that exhibit the same failure output (the same error type and error message) and look at the number of the tests in the same group (the $max\_output\_clus\_size$ in Line 8). This is based on the intuition that, if multiple tests show similar failure behavior, then it is likely that the LLM is 'confident' in its predictions as its independent predictions 'agree' with each other, and there is a good chance that bug reproduction has succeeded. While the similarity of tests is also related to the amount of randomness in sampling, controlled by the sampling temperature parameter of LLMs, we find that as long as the LLM is not completely deterministic (i.e. temperature is zero), the amount of consistency is predictive of bug reproduction performance in Section 6.5.2. This is a well-known and universal property of LLMs, also explored by Wang et al. [11] and verified in our experiments. LIBRO can be configured to only show results when there is significant agreement in the output (setting the agreement threshold Thr high) or show more exploratory results (setting Thr low).

Once it decides to show its results, LIBRO relies on three heuristics to rank generated tests, in ascending order of discriminative strength. First, tests are likely to be bug reproducing if the fail message and/or the test code shows the behavior (exceptions or output values) observed and mentioned in the bug report. While this heuristic is precise, its decisions are not very discriminative, as tests can only be divided into groups of 'contained' versus 'not contained'. Next, we look at the 'agreement' between generated tests by looking at output cluster size ($output\_clus\_size$), which represents the 'consensus' of the LLM generations. Finally, LIBRO prioritizes based on test length (as shorter tests are easier to understand), which is the finest-grained signal. We first leave only syntactically unique tests (Line 11), then sort output clusters and tests within those clusters using the heuristics above (Lines 16 and 18).

Algorithm 2: Test Selection and Ranking

```
   Input: Pairs of modified test suites and injected test methods
          $\mathcal{S}_{\mathcal{T}'}$; target program with bug $P_b$; bug report $BR$;
          agreement threshold $Thr$;
   Output: Ordered list of ranked tests $ranking$;
1  $FIB \leftarrow$ set();
2  for $(\mathcal{T}', tm_i) \in \mathcal{S}_{\mathcal{T}'}$ do
3      $r \leftarrow$ executeTest($\mathcal{T}', P_b$);
4      if hasNoCompileError($r$) && isFailed($tm_i, r$) then
5          $FIB$.add(($tm_i, r$));
6  $clusters \leftarrow$clusterByFailureOutputs($FIB$);
7  $output\_clus\_size \leftarrow clusters$.map(size);
8  $max\_output\_clus\_size \leftarrow$max($output\_clus\_size$);
9  if $max\_output\_clus\_size \leq Thr$ then
10     return list();
11 $FIB_{uniq} \leftarrow$ removeSyntacticEquivalents($FIB$);
12 $br\_output\_match \leftarrow clusters$.map(matchOutputWithReport($BR$)
    );
13 $br\_test\_match \leftarrow FIB_{uniq}$.map(matchTestWithReport($BR$));
14 $tok\_cnts \leftarrow FIB_{uniq}$.map(countTokens);
15 $ranking \leftarrow$list();
16 $clusters \leftarrow clusters$.sortBy(
    $br\_output\_match, output\_clus\_size, tok\_cnts$);
17 for $clus \in clusters$ do
18     $clus \leftarrow clus$.sortBy($br\_test\_match, tok\_cnts$);
19 for $i = 0; i <$max($output\_clus\_size$); $i \leftarrow i + 1$ do
20     for $clus \in clusters$ do
21         if $i < clus$.length() then
22             $ranking$.push($clus$[i])
23 return $ranking$;
```

As tests with the same failure output are similar to each other, we expect that, if one test from a cluster is not BRT, the rest from the same cluster are likely not BRT as well. Hence, LIBRO shows tests from a diverse array of clusters. For each ith iteration in Line 19-22, the ith ranked test from each cluster is selected and added to the list.

4 EVALUATION

This section provides evaluation details for our experiments.

4.1 Dataset

To evaluate LIBRO, and its performance when using different LLMs, over a large number of real-world bugs, we employ the Defects4J v2.0 dataset, which is a collection of Java bugs with bug reports[2] collected from open-source repositories. Consequently, the dataset provides a convenient means of evaluating how many bugs LIBRO could reproduce over

[2] Except for the Chart project, for which only 8 bugs have reports

Organization	LLM	Accessible	Size	Downloadable	Code Model	Release Year	Chat
OpenAI	code-davinci-002 (Codex)	Not Since Mar. 2023	176B?	No	Yes	2021	No
	text-davinci-003	No	176B?	No	No	2022	No
	gpt-3.5-turbo-0301	Yes	?	No	No	2023	Yes
	gpt-3.5-turbo-0613	Yes	?	No	No	2023	Yes
BigScience	Bloom	Yes	176B	Yes	No	2022	No
	BloomZ	Yes	176B	Yes	No	2022	No
Meta	Incoder	Yes	(1, 6)B	Yes	Yes	2022	No
Salesforce	CodeGen2	Yes	(1, 3.7, 7, 16)B	Yes	Yes	2023	No
BigCode	StarCoder	Yes	15B	Yes	Yes	2023	No
	StarCoderBase	Yes	15B	Yes	Yes	2023	No
	StarCoderPlus	Yes	15B	Yes	Yes	2023	No

TABLE 3: LLMs used in our evaluation.

different projects. While there are 814 bugs in the Defects4J v2.0 dataset, 58 bugs were excluded as upon inspection their report content did not match the patch; furthermore, six bugs were excluded as they had a different directory structure between the buggy and fixed versions, making it difficult to consistently apply our evaluation pipeline. Thus, we evaluated over **750** bugs that had a good report-bug matching, and for which there was not substantial refactoring between the buggy and fixed versions. To compare against the state-of-the-art crash reproduction technique EvoCrash [25], we used the 60 bugs in Defects4J that were included in the JCrashPack [15] dataset. For reference, in the entire 750-bug dataset that we used, 248 (33.1%) were crash bugs, in which the 60 bugs in the JCrashPack dataset are included.

The Defects4J dataset is likely in most code-based LLM training data, as was argued by the recent work of Lee et al. [26], even if the prompt format we use is unlikely to have appeared verbatim in the data. Indeed, Lee et al. note that specifically for StarCoder, many of the Defects4J bug-reproducing tests are included in the StarCoder pretraining data. This threatens conclusions derived from Defects4J, as it is unclear whether any results are due to LLM memorization or are genuinely likely to be reproducible by practitioners in their own projects. To mitigate such concerns, from 17 GitHub repositories[3] that use JUnit, we gather 581 Pull Requests (PR) created after the Codex training data cutoff point of July 2021 [27], ensuring that this dataset could not have been used to train Codex. We further check if a PR adds any test to the project (435 left after discarding non-test-introducing ones), and filter out the PRs that are not merged to the main branch or associated with multiple issues - this leaves 84 PRs.

As a final step, we verified that each PR added a bug-reproducing test by verifying that the developer-added test in the PR failed on the pre-merge commit of the project, but passed on the post-merge commit. PRs that failed to introduce any such tests were discarded. This left us with a final dataset of **31** bugs which could be reproduced, and their corresponding bug reports. In the remainder of this paper, we refer to this dataset as the GHRB (GitHub Recent Bugs) dataset; we use this dataset to verify that the results we obtain for Defects4J are not simply due to LLM verbatim memorization of training data. While this dataset ensures that the developer-written bug-reproducing tests were not in the training data of the LLMs, it does not exclude the possibility that an earlier version of the project source code was not in the training data. Further, as a result of the existence of bug-reproducing tests, all bug reports used in our study will describe functional bugs, rather than non-functional bugs.

4.2 Metrics

The goal of LIBRO is to generate Bug Reproducing Tests (BRTs) for each bug. We treat a generated test as a BRT if and only if it (i) fails on the initially buggy version of the code and (ii) passes on the fixed version of the code. For a bug to be *reproduced* by LIBRO, one or more BRTs must be generated for the bug. For each setting and technique, the number of BRTs is counted.

To maximize the realism of our evaluation, when using the Defects4J benchmark, we use the `PRE_FIX_REVISION` and `POST_FIX_REVISION` versions of the code, which reflect the unaltered state of the project when a bug was reported/fixed. When using the GHRB suite of bugs, we use the commits before and after a pull request as buggy and fixed versions.

As a baseline, we compare against the EvoCrash [25] technique. By default, EvoCrash will only check whether the crash stack is reproduced, instead of verifying over the fixed version. To make comparison consistent, we apply our definition of BRT, and execute EvoCrash tests on the buggy and fixed versions to determine success.

As LIBRO not only produces BRTs, but also selects results and BRTs, evaluation metrics are required for this aspect as well. For result selection, we use the ROC-AUC metric, which evaluates the performance of a classifier regardless of the specific threshold used. To evaluate the rankings, we use the $acc@n$ and wef metrics. $acc@n$ measures the number of bugs for which LIBRO could successfully rank a BRT within the top n suggestions. wef measures the amount of wasted developer effort when inspecting the suggested tests from LIBRO; a variant of it, $wef@n$, measures the wasted effort when inspecting the top n candidates.

4.3 Environment

All test execution experiments were performed on a machine running Ubuntu 18.04.6 LTS, with 32GB of RAM and

[3] These repositories have been manually chosen from either Defects4J projects that are on GitHub and open to new issues, or Java projects that have been modified since 10th July 2022 with at least 100 or more stars, as of 1st of August 2022. A list of the 17 repositories is available in the artifact of our prior work [10].

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3450837

Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz CPU. Meanwhile, we set up a server for hosting open-source LLM inference, which ran Ubuntu 20.04.6 LTS and was equipped with 16 Intel(R) Xeon(R) Gold 5222 CPU @ 3.80GHz CPUs and four NVIDIA RTX 3090 GPUs, with a total VRAM of 96GB. This was enough to run the 15B StarCoder LLM which has 8,000 context length without weight quantization. The server was implemented to behave similarly to the OpenAI API, with configurable LLMs being loaded as needed to run the experiments.

Regarding the LLMs we use, we present their characteristics in Table 3. For the closed-source LLMs from OpenAI, we access them via the public API, and evaluate the performance of the `gpt-3.5-turbo-0301` and `gpt-3.5-turbo-0613` chat-based models as well as the autocompletion-based `text-davinci-003` model, and compare the bug reproduction results with the results of using the `code-davinci-002` (Codex) model from our prior work [10]. For the open-source LLMs, we evaluate the BLOOM [20]/BLOOMZ [28] family of LLMs, which were the largest open-source LLMs at the time of experimentation; thus, we decided to experiment with them, even though they were not specifically trained on code. As our infrastructure could not host the BLOOM LLMs, we used the HuggingFace inference API to conduct our experiments. All the other LLMs that we used were trained on code. The InCoder [29] family of LLMs consists of two LLMs of different sizes, and are code-based LLMs trained by what is now Meta, with the largest model having 6 billion parameters. The CodeGen2 [30] family of LLMs consisting of four LLMs of different sizes, was trained by Salesforce, with the largest model having 16 billion parameters. Finally, there were three variants of the StarCoder [12] model; the initial StarCoderBase model which was trained on a large corpus (more than one trillion tokens) of source code, the StarCoder model which is a fine-tuned version of StarCoderBase on Python code, and the StarCoderPlus model which was further fine-tuned from the StarCoder model so that it could operate in a chat format. We utilize the variants of the open-source models to discuss the effect of training data and model size on bug reproduction performance.

For all models, we set the sampling temperature to 0.7 (except in Section 6.4.5 and Section 6.5.2 where we explicitly experiment with the temperature parameter), and the maximum number of generated tokens to 256. We do so as previous work has found LLMs tend to yield good performance in when the temperature is between 0.6 and 0.8 [7], [31]. When comparing settings and LLMs in Defects4J, by default we sample 10 tests (denoted by n=10), due to computational constraints; however, for the Codex (code-davinci-002) and StarCoder LLM, we experiment with sampling 50 tests as well. For the GHRB dataset, due to the small number of bugs, we consistently evaluate with n=50 tests. We script our experiments using Python 3.9, and parse Java files with the `javalang` library [32]. Our tool is public and archived[4], and the replication package for our journal extension is also available as well[5].

5 Research Questions

This section presents our research questions. In RQ1-3, we present the effectiveness of Libro when using the Codex (`code-davinci-002`) LLM from OpenAI. In RQ3 and RQ4, we use different LLMs and LLM settings for Libro and compare their efficacy.

5.1 RQ1: Efficacy

In RQ1, the bug-reproducing performance of Libro when using Codex is evaluated and compared to baselines, based on the Defects4J dataset.

- **RQ1-1: How many bug reproducing tests can Libro generate?** We present how many bugs Libro could reproduce based on different prompts, when using the best-performing LLM, `code-davinci-002`.
- **RQ1-2: How does Libro compare to other techniques?** We compare against baselines to further demonstrate the performance of Libro. As there are no general bug reproduction techniques that we are aware of, we compare against the state-of-the-art crash reproduction technique EvoCrash. Furthermore, we implement and compare with a 'Copy&Paste' baseline which directly uses code snippets from the bug report, extracted via infoZilla [33] or by the HTML `<pre>` tag, as bug-reproducing tests.

5.2 RQ2: Efficiency

RQ2 inspects various aspects of the efficiency of Libro, so that practitioners may estimate the costs of deploying Libro on new projects.

- **RQ2-1: How many LLM queries are required?** The number of queries to Codex required to reach a certain number of bugs reproduced on the Defects4J dataset is evaluated.
- **RQ2-2: How much time does Libro need?** The amount of computation time used at each stage of the Libro pipeline is evaluated.
- **RQ2-3: How many tests should the developer inspect?** To estimate the degree of developer effort needed to use Libro, the number of bugs for which tests would be suggested, and the number of tests developers would need to inspect, is evaluated.

5.3 RQ3: Practicality

RQ3 investigates how well Libro with Codex generalizes, by evaluating it on the GHRB dataset instead of the Defects4J dataset, which may be part of Codex training data.

- **RQ3-1: How often can Libro reproduce bugs in the wild?** The number of bugs that can be reproduced by Libro on the GHRB dataset is evaluated, mitigating data leakage risks.
- **RQ3-2: How reliable are the selection and ranking techniques of Libro?** Whether the selection and ranking algorithm work on an unseen dataset is evaluated, demonstrating that the pipeline captures generalizable properties of Codex.

[4] https://doi.org/10.6084/m9.figshare.26095819.v1
[5] https://github.com/coinse/libro-journal-artifact

5.4 RQ4: Sensitivity to LLMs

With RQ4, we present the results of a large-scale study involving 15 LLMs, and present how the performance of LIBRO changes as the underlying LLM varies.

- **RQ4-1: What is the best LLM to use given GPU limits?** We present the results of evaluating LIBRO with multiple different LLMs, and propose which LLM performs best when GPU memory is restricted.
- **RQ4-2: How well do the LLMs perform on holdout bugs?** We evaluate LIBRO with different LLMs on the holdout GHRB bug dataset to see whether the LLMs used by LIBRO are simply relying on dataset memorization.
- **RQ4-3: How has ChatGPT's behavior changed over time?** We compare the performance of LIBRO when different ChatGPT models are used as the LLM, similarly to Chen et al. [18], and analyze why such a change occurred.
- **RQ4-4: How does LLM size influence performance within the same LLM family?** By comparing LLMs from the same family, which were trained with the same data and techniques with the only difference being the model size, we compare how LIBRO performance changes as LLM size is increased.
- **RQ4-5: How does LLM sampling temperature influence performance?** LLMs have a 'temperature' parameter that controls the degree of randomness in the outputs of LLMs. The influence the temperature parameter has on the performance of LIBRO-StarCoder is evaluated.

5.5 RQ5: Sensitivity to LLM Self-Consistency

The selection and ranking algorithm of LIBRO, while developed independently, shares the same intuition as the phenomenon in LLMs known as self-consistency [11]. Consequently, the selection and ranking algorithm of LIBRO is a good way of evaluating how consistent the self-consistency phenomenon is over different LLMs.

- **RQ5-1: Does the choice of LLM influence the performance of LIBRO postprocessing?** Different LLMs may have different self-consistency characteristics, and consequently the selection and ranking algorithm of LIBRO may show different performance. To evaluate this, we use the results of multiple LLMs from RQ4 and evaluate how well our selection and ranking algorithm works.
- **RQ5-2: Does the sampling temperature of LLM change the performance of LIBRO postprocessing?** The sampling temperature of LLMs has an even more direct relationship to self-consistency – when the temperature is low, more results will be similar and vice versa. To this end, we evaluate what influence sampling temperature has on our selection and ranking algorithm.

6 EXPERIMENTAL RESULTS

6.1 RQ1. How effective is LIBRO?

6.1.1 RQ1-1

Table 4 shows which prompt/information settings work best, where $n = N$ means we queried the `code-davinci-002` N times for reproducing tests. When using examples from the source project, we use the oldest tests available within that project; otherwise, we use two handpicked report-test pairs (Time-24, Lang-1) throughout all projects. We find that providing constructors (*à la* AthenaTest [34]) does not help significantly, but adding stack traces does help reproduce crash bugs, indicating that LIBRO can benefit from using the stack information to replicate issues more accurately. Interestingly, adding within-project examples shows poorer performance: inspection of these cases has revealed that, in such cases, LIBRO simply copied the provided example even when it should not have, leading to lower performance. We also find that the number of examples makes a significant difference when comparing no example results to one example or two example results, confirming the existing finding that adding examples helps improve performance. Furthermore, the number of times the LLM is queried also influences performance, as we further explore in RQ2-1. As the two-example n=50 setting shows the best performance, we use it as the default setting throughout the rest of the paper.

TABLE 4: Reproduction performance for different prompts

Setting	reproduced	FIB
No Example (n=10)	124	440
One Example (n=10)	166	417
One Example from Source Project (n=10)	152	455
One Example with Constructor Info (n=10)	167	430
Two Examples (n=10, 5th percentile)	161	386
Two Examples (n=10, median)	173	409
Two Examples (n=10, 95th percentile)	184	429
Two Examples (n=50)	**251**	**570**
One Example, Crash Bugs (n=10)	69	153
One Example with Stack, Crash Bugs (n=10)	84	155

Under the two-example n=50 setting, we find that overall **251** bugs, or 33.5% of 750 studied Defects4J bugs, are reproduced by LIBRO. Table 5 presents a breakdown of the performance per project. While there is at least one bug reproduced for every project, the proportion of bugs reproduced can vary significantly. For example, LIBRO reproduces a small number of bugs in the Closure project, which is known to have a unique test structure [35]. On the other hand, the performance is stronger for the Lang or Jsoup projects, whose tests are generally self-contained and simple. Additionally, we find that the average length of the generated test body is about 6.5 lines (excluding comments and whitespace characters), indicating LIBRO is capable of writing meaningfully long tests.

> **Answer to RQ1-1:** A large (251) number of bugs can be replicated automatically using Codex, with bugs replicated over a diverse group of projects. Further, the number of examples in the prompt and the number of generation attempts have a strong effect on performance.

TABLE 5: Bug reproduction per project in Defects4J: x/y means x reproduced out of y bugs

Project	rep/total	Project	rep/total
Chart	5/7	JacksonDatabind	30/107
Cli	14/29	JacksonXml	2/6
Closure	2/172	Jsoup	56/92
Codec	10/18	JxPath	3/19
Collections	1/4	Lang	46/63
Compress	4/46	Math	43/104
Csv	6/16	Mockito	1/13
Gson	7/11	Time	13/19
JacksonCore	8/24	**Total**	**251/750**

6.1.2 RQ1-2

We further compare LIBRO against the state-of-the-art crash reproduction technique, EvoCrash, and the 'Copy&Paste baseline' that uses code snippets from the bug reports. We present the comparison results in Figure 2. We find LIBRO replicates a large and distinct group of bugs compared to other baselines. LIBRO reproduced 91 more unique bugs (19 being crash bugs) than EvoCrash, which demonstrates that LIBRO can reproduce non-crash bugs prior work could not handle (Fig. 2(b)). On the other hand, the Copy&Paste baseline shows that, while the BRT is sometimes included in the bug report, the report-to-test task is not at all trivial. Interestingly, eight bugs reproduced by the Copy&Paste baseline were not reproduced by LIBRO; we find that this is due to long tests that exceed the generation length of LIBRO, or due to dependency on complex helper functions.

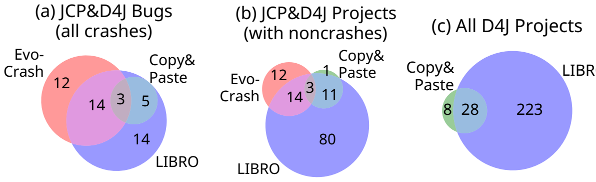


Fig. 2: Baseline comparison on bug reproduction capability

Answer to RQ1-2: LIBRO is capable of replicating a large and distinct group of bugs relative to prior work.

6.2 RQ2. How efficient is LIBRO?

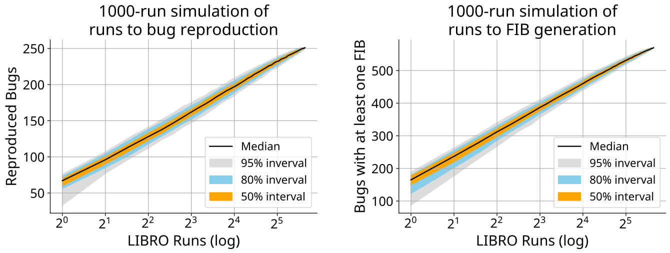


Fig. 3: Generation attempts to performance. Left depicts bugs reproduced as attempts increase, right for FIB

6.2.1 RQ2-1

Here, we investigate how many tests must be generated to attain a certain bug reproduction performance. To do so, for each Defects4J bug, we randomly sample x tests from the 50 generated under the default setting, leaving a reduced number of tests per bug. We then check the number of bugs reproduced y when using only those sampled tests. We repeat this process 1,000 times to approximate the distribution.

The results are presented in Figure 3. Note that the x-axis is in log scale. Interestingly, we find a logarithmic relation holds between the number of test generation attempts and the median bug reproduction performance. This suggests that it becomes increasingly difficult, yet stays possible, to replicate more bugs by simply generating more tests. As the graph shows no signs of plateauing, experimenting with an even greater sample of tests may result in better bug reproduction results.

Answer to RQ2-1: The number of bugs reproduced increases logarithmically to the number of tests generated; we did not reach a plateau in our experiments.

TABLE 6: The time required for the pipeline of LIBRO

	Prompt	API	Processing	Running	Ranking	Total
Single Run	<1 μs	5.85s	1.23s	4.00s	-	11.1s
50-test Run	<1 μs	292s	34.8s	117s	0.02s	444s

6.2.2 RQ2-2

We report the time it takes to perform each step of our pipeline in Table 6. We find API querying takes the greatest amount of time, requiring about 5.85 seconds. Postprocessing and test executions take 1.23 and 4 seconds per test (when the test executes), respectively. Overall, LIBRO took an average of 444 seconds to generate 50 tests and process them, which is well within the 10-minute search budget often used by search-based techniques [15].

Answer to RQ2-2: Our time measurement suggests that LIBRO does not take a significantly longer time than other methods to use.

6.2.3 RQ2-3

With this research question, we measure how effectively LIBRO prioritizes bug reproducing tests via its selection and ranking procedure. As LIBRO only shows results above a certain agreement threshold, Thr from Section 3.4, we first present the trade-off between the number of total bugs reproduced and precision (i.e., the proportion of successfully reproduced bugs among all selected by LIBRO) in Figure 4. As we increase the threshold, more suggestions (including BRTs) are discarded, but the precision gets higher, suggesting one can smoothly increase precision by tuning the selection threshold.

We specifically set the agreement threshold to 1, a conservative value, in order to preserve as many reproduced bugs as possible. Among the 570 bugs with a FIB, 350 bugs are selected. Of those 350, 219 are reproduced (leading to a precision of $0.63(=\frac{219}{350})$ whereas recall (i.e., proportion of selected reproduced bugs among all reproduced bugs) is $0.87(=\frac{219}{251})$. From the opposite perspective, the selection process filters out 188 bugs that were not reproduced, while dropping only a few successfully reproduced bugs. Note

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3450837

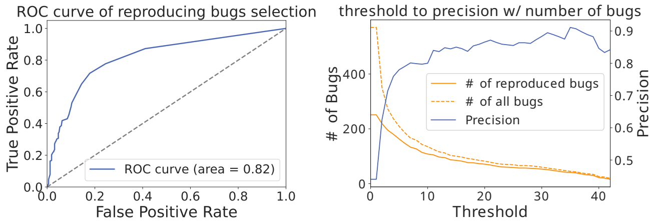


Fig. 4: ROC curve of bug selection (Left), Effect of thresholds to the number of bugs selected and precision (Right)

TABLE 7: Ranking Performance Comparison between LIBRO and Random Baseline

	Defects4J				GHRB			
	$acc@n$ $(precision)$		$wef@n_{agg}$		$acc@n$ $(precision)$		$wef@n_{agg}$	
n	LIBRO	random	LIBRO	random	LIBRO	random	LIBRO	random
1	**149 (0.43)**	116 (0.33)	**201 (0.57)**	234 (0.67)	**6 (0.29)**	4.8 (0.23)	**15 (0.71)**	16.2 (0.77)
3	**184 (0.53)**	172 (0.49)	**539 (1.54)**	599 (1.71)	**7 (0.33)**	6.6 (0.31)	**42 (2.0)**	44.6 (2.12)
5	**199 (0.57)**	192 (0.55)	**797 (2.28)**	874 (2.5)	**8 (0.38)**	7.3 (0.35)	**60 (2.86)**	64.3 (3.06)

that if we set the threshold to 10, a more aggressive value, we can achieve a higher precision of 0.84 for a recall of 0.42. In any case, as Figure 4 presents, our selection technique is significantly better than random, indicating it can save developer resources.

Among the selected bugs, we assess how effective the test rankings of LIBRO are over a random baseline. The random approach randomly ranks the syntactic clusters (groups of syntactically equivalent FIB tests) of the generated tests. We run the random baseline 100 times and average the results.

Table 7 presents the ranking evaluation results. On the Defects4J benchmark, the ranking technique of LIBRO improves upon the random baseline across all of the $acc@n$ metrics, presenting 30, 14, and 7 more BRTs than the random baseline on $n = 1$, 3, and 5 respectively. Regarding $acc@1$, the first column shows that 43% of the top ranked tests produced by LIBRO successfully reproduce the original bug report on the first try. When n increases to 5, BRTs can be found in 57% of the selected bugs, or 80% of all reproduced bugs. The conservative threshold choice here, emphasizes recall over precision. However, if the threshold is raised, the maximum precision can rise to 0.8 (for $Thr = 10, n = 5$).

The $wef@n_{agg}$ values are additionally reported by both summing and averaging the $wef@n$ of all (350) selected bugs. The summed $wef@n$ value indicates the total number of non-BRTs that would be manually examined within the top n ranked tests. Smaller $wef@n$ values indicate that a technique delivers more bug reproducing tests. Overall, the ranking of LIBRO saves up to 14.5% of wasted effort when compared to the random baseline, even after bugs are selected. Based on these results, we conclude that LIBRO can reduce wasted inspection effort and thus be useful to assist developers.

> **Answer to RQ2-3:** LIBRO can reduce both the number of bugs and tests that must be inspected: 33% of the bugs are safely discarded while preserving 87% of the successful bug reproduction. Among selected bug sets, 80% of all bug reproductions can be found within 5 inspections.

6.3 RQ3. How well would LIBRO work in practice?

TABLE 8: Bug Reproduction in GHRB: x/y means x reproduced out of y bugs

Project	rep/total	Project	rep/total	Project	rep/total
AssertJ	3/5	Jackson	0/2	Gson	4/7
checkstyle	0/13	Jsoup	2/2	sslcontext	1/2

6.3.1 RQ3-1

We explore the performance of LIBRO when operating on the GHRB dataset of recent bug reports. We find that of the 31 bug reports we study, LIBRO can automatically generate bug reproducing tests for 10 bugs based on 50 trials, for a success rate of **32.2%**. This success rate is similar to the results from Defects4J presented in RQ1-1, suggesting LIBRO generalizes to new bug reports. A breakdown of results by project is provided in Table 8. Bugs are successfully reproduced in AssertJ, Jsoup, Gson, and sslcontext, while they were not reproduced in the other two. We could not reproduce bugs from the Checkstyle project, despite it having a large number of bugs; upon inspection, we find that this is because the project's tests rely heavily on external files, which LIBRO has no access to, as shown in Section 7.1. LIBRO also does not generate BRTs for the Jackson project, but the small number of bugs in the Jackson project make it difficult to draw conclusions from it.

> **Answer to RQ3-1:** LIBRO is capable of generating bug reproducing tests even for recent data, suggesting it is not simply remembering what it trained with.

6.3.2 RQ3-2

LIBRO uses several predictive factors correlated with successful bug reproduction for selecting bugs and ranking tests. In this research question, we check whether the identified patterns based on the Defects4J dataset continue to hold in the recent GHRB dataset.

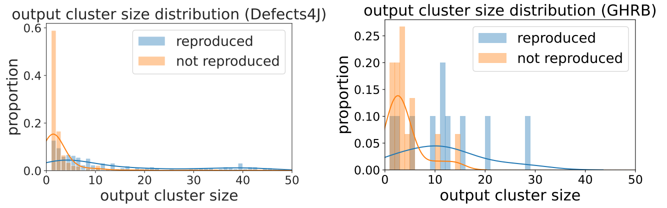


Fig. 5: Distribution of the $max\_output\_clus\_size$ values for reproduced and not-reproduced bugs

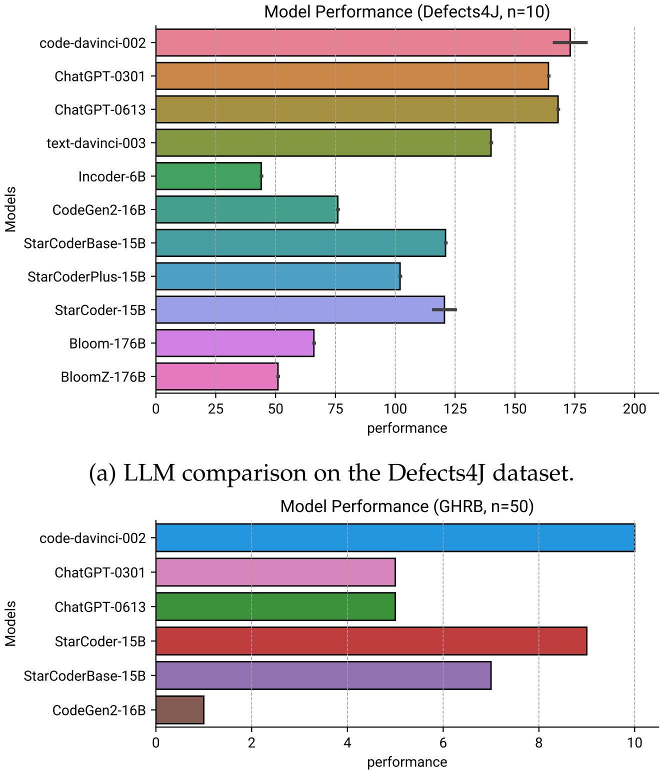
Recall that we use the maximum output cluster size as a measure of agreement among the FIBs, and thus as a selection criterion to identify whether a bug has been reproduced. To observe whether the criterion is a reliable indicator to predict the success of bug reproduction, we observe the trend of $max\_output\_clus\_size$ between the two datasets, with and without BRTs. In Figure 5, we see that the bugs with no BRT typically have small $max\_output\_clus\_size$, mostly under ten; this pattern is consistent in both datasets.

The ranking results of GHRB are also presented in Table 7. They are consistent to the results from Defects4J, indicating the features used for our ranking strategy continue to be good indicators of successful bug reproduction.

> **Answer to RQ3-2:** The factors used for the ranking and selection of LIBRO consistently predict bug reproduction in real-world data.

6.4 RQ4. How does the choice of LLM influence LIBRO performance?

6.4.1 RQ4-1



(a) LLM comparison on the Defects4J dataset.

(b) LLM comparison on the GHRB dataset.

Fig. 6: LLM to reproduction performance, with the upper graph depicting Defects4J performance, and the right graph depicting GHRB performance.

While the previous research questions focused on the performance of LIBRO when using one of the first code-based LLMs (Codex), subsequent results evaluate how much LIBRO performance changes when using LLMs of different sizes and training regimes, with a particular focus on open-source LLMs that are available to academia. While some work has evaluated the performance of open-source LLMs [36], since then, multiple open-source LLMs have been introduced thanks to the increased interest in LLMs [12], [30], [37], [38], which to the best of our knowledge have not yet been evaluated in the software engineering literature.

Figure 6 shows the performance of multiple LLMs when given two examples in the prompt. The best performing LLM was Codex, which could reproduce a median of 173 bugs given 10 test generation attempts for Defects4J dataset. Comparing the performance of the open-source models, we find that the StarCoder family of models shows the best performance, with the best model, StarCoder-15B, being capable of reproducing 125 bugs, or 73% of the performance of Codex. This approaches the results of the currently available OpenAI models, and could provide stable performance, unlike using the ChatGPT models, which may yield different results without notice. Comparing the StarCoder family of models, although StarCoder is a fine-tuned version of StarCoderBase trained on Python, this training did not degrade performance when reproducing Java bugs, indicating that fine-tuning on one language does not necessarily hurt performance on other languages. Meanwhile, StarCoderPlus, which is a fine-tuned version of StarCoder trained on natural language, showed a substantially worse performance. This can also be seen in the Bloom family of models: the fine-tuned BloomZ model performed substantially worse on reproduction than the original Bloom model. Overall, we find that the open-source LLMs can also reproduce a substantial number of bugs (albeit lower than the OpenAI models), and thus are a viable option when security is a greater concern than performance. As StarCoder showed the best performance among the open-source LLMs, we use StarCoder for the subsequent experiments on temperature.

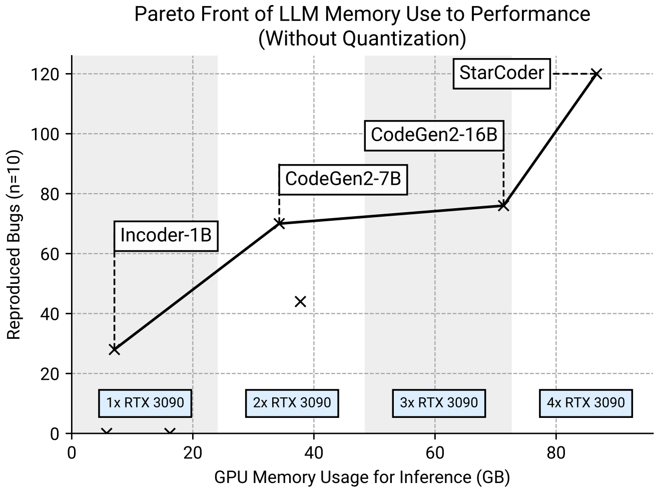


Fig. 7: LLM GPU memory use to performance. The graph shows the Pareto front of performance between memory use and performance.

By combining this data with our measurement of the memory consumption of each model, we plot the tradeoff between GPU memory usage and performance in Figure 7. As expected, there is a trend of better performance as more GPU memory is used. The four models on the Pareto front (Incoder-1B, CodeGen2-7B, CodeGen2-16B, StarCoder) are highlighted as well; these are models such that their performance is nonzero, and there is no model that both performs better and uses less memory. Conveniently, each model can be mapped to a different number of GPUs as well as Figure 7 shows. This information could be helpful to practitioners/researchers when they make decisions on which LLM to deploy based on their GPU situation.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3450837

TABLE 9: OpenAI model performance under prompts

Model Prompt	GPT-0301 Prompt 1	GPT-0613 Prompt 1	GPT-0613 Prompt 2
Performance	164	72	168

Answer to RQ4-1: The performance of LIBRO is influenced significantly by the LLM used. Codex shows the best performance of all LLMs, and StarCoder shows the best performance among the open-source LLMs. With less GPUs, Incoder-1B or CodeGen2-7B models are good options as well.

6.4.2 RQ4-2

To evaluate the performance of LIBRO on the holdout bugs of the GHRB dataset and thus confirm that the LLMs are not simply repeating training data, we select six models that showed strong performance on Defects4J: Codex, GPT-3.5-0301, GPT-3.5-0613, StarCoder, StarCoderBase, and CodeGen2 models. As explained in Section 4.1, we collected recent bug report and reproducing test pairs after the Codex training data cutoff date, but we find that all the reproducing tests contained in GHRB do not belong to the Stack dataset, which is used to train StarCoder family and CodeGen2 models. We derive this observation from StarCoder's dataset membership test[6] provided along with the dataset themselves, a finding also supported by Lee et al. [26].

We generated 50 tests for each bug report with LIBRO for all five models, and the results are presented in Figure 6. The trends of performance observed in Defects4J are observed in GHRB as well, with StarCoder performing the best after code-davinci-002, achieving 90% of the performance of code-davinci-002 when generating 50 tests. Overall, the average reproduction ratio of StarCoder is 25.2% for Defects4J and 29.0% for GHRB in the setting of generating 50 tests, while the GPT-3.5 models reproduced about 22% of bug reports from Defects4J and 16.1% of GHRB when generating 50 tests. As LLMs tend to show similar performance for the GHRB data which is likely not part of the training dataset of any LLM, we suggest that LIBRO with general LLMs can be used for novel bug reproduction.

Answer to RQ4-2: LLMs can still perform well for held-out bugs; similarly to Defects4J, StarCoder shows the best performance among the open-source models.

6.4.3 RQ4-3

While the LLMs of OpenAI are the most well-known and show strong performance on a multitude of tasks [6], there are few details known about the models, particularly starting with the most recent model, GPT-4 [39], which did not provide even basic details about the model such as model size. Furthermore, OpenAI LLMs are regularly updated, and thus pose a challenge for reproducibility in academic research. For example, the LLM that was used in our initial experiments, Codex (code-davinci-002), has since become inaccessible to the public. Although OpenAI recently announced a new feature for setting a seed to achieve deterministic outputs [40], it remains sensitive to internal system updates.

Comparing the OpenAI LLM models, gpt-3.5-turbo-0301 achieved a similar performance of 164 bugs given 10 test generation attempts, but gpt-3.5-turbo-0613 achieved a much worse performance than both of these models, only reproducing 72 bugs under the same condition, as shown in Table 9. Initially, such results may appear to represent a shift in model performance, as has been suggested by Chen et al. [18] which noted that the number of executable Python scripts generated by ChatGPT had reduced. Inspecting the results from gpt-3.5-turbo-0613, we find that gpt-0613 would generate full test files instead of test methods, so that the generated code could no longer be processed correctly by our postprocessing pipeline. Modifying the prompt by placing the examples in the system message and emphasizing the need to generate test methods instead of test files, gpt-3.5-turbo-0613 could achieve similar performance to its earlier version. Thus, it is difficult to conclude from our data that ChatGPT has become "worse" over time, as Chen et al. [18] argue. Rather, as noted by Narayanan and Kappor [41], it highlights the risk when building services on top of ChatGPT: its behavior can change at any time, and thus postprocessing pipelines or prompts may need to adapt without warning. This also shows the importance of appropriate prompting for LLMs, as the prompt used can affect the performance substantially.

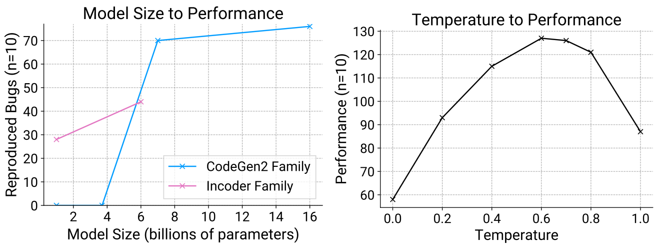
Answer to RQ4-3: Similarly to prior work, we observe a change in ChatGPT behavior; in our case, ChatGPT became less susceptible to few-shot learning, and our post-processing pipeline which relied on a specific output format failed.

6.4.4 RQ4-4

While Figure 6 compared the performance of LLMs trained in different ways, we also make a comparison between LLMs that are from the same family and were thus trained in a similar manner, but are of substantially different size, to demonstrate how LLM size can affect bug reproduction performance. We plot the results of these experiments in Figure 8a. As the graph shows, bug reproduction suddenly becomes possible when using the 7B model for CodeGen2. Such results are reminiscent of 'emergent' properties of LLMs [42], in which LLM capabilities suddenly appear at a certain model size, which makes LLM capabilities difficult to predict prior to training. On the other hand, in the Incoder family, even the 1B model can reproduce a certain amount of bugs using our default prompt. Regardless of whether the property is emergent, the results in Figure 6 show that bug reproduction performance tends to increase as model size increases.

Answer to RQ4-4: LIBRO performance improves as the underlying LLM size increases; for CodeGen2, a sudden appearance of reproduction capability is observed.

[6] https://stack.dataportraits.org/



(a) Model Size

(b) Temperature

Fig. 8: Evaluation of the influence of LLM configuration to performance.

6.4.5 RQ4-5

Figure 8b shows the performance of Libro when using StarCoder. As the graph shows, we find that the performance was best when the temperature was 0.6, which was similar to our initial setting of temperature=0.7. At temperature=0.6, Libro-StarCoder could reproduce 127 bugs when generating ten tests for each bug report. Looking at each temperature, we find that at temperatures lower than 0.6, the LLM tends to generate identical or similar tests for a given bug report, and thus does not reproduce more bugs as more tests are generated. Meanwhile, for higher temperatures, the coherence of the LLM-generated results deteriorates, and thus increasingly less bugs are reproduced. Indeed, while not shown in the graph, our experiments when the temperature was set to 2.0 revealed that the LLM would almost exclusively generate unparsable code, indicating that setting the LLM to the right temperature is important when achieving strong bug reproduction performance.

Answer to RQ4-5: The performance of Libro-StarCoder is optimized when the temperature is set to 0.6, which gets a good balance between generation diversity and result coherence.

6.5 RQ5. How does the choice of LLM influence the selection and ranking aspect of Libro?

6.5.1 RQ5-1

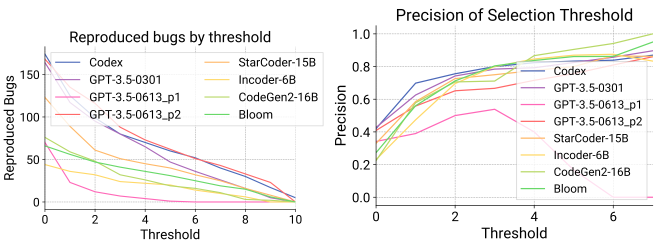


Fig. 9: Number of reproduced bugs and precision by selection with varying thresholds.

As Libro selects bugs and prioritizes tests to reduce the number of bugs and tests that need to be inspected, we further investigate how different LLMs perform when using the selection and ranking strategy. We selected the best-performing models from each model family and focused our evaluation on those models for simplicity.

Figure 9 shows that the number of reproduced bugs and precision (i.e., the number of reproduced bugs among the selected ones divided by the number of selected bugs) with varying thresholds. As the threshold rises, the number of reproduced bugs decrease, while precision generally increases (with the exception of gpt-3.5-turbo-0613). Based on the result, we suggest that Libro's selection criterion, maximum size of output clusters, is a useful indicator for successful bug reproduction regardless of the LLM used. In addition, the abnormally low performance of gpt-3.5-turbo-0613 when using Prompt 1 (Figure 9, GPT-3.5-0613_p1; for discussion, see Section 6.4.4) is accompanied by abnormal threshold-precision behavior, in which increasing the selection threshold does not lead to improved precision. Such abnormalities may potentially used as diagnostics for LLM-based applications, which may provide insight that raw performance may not provide.

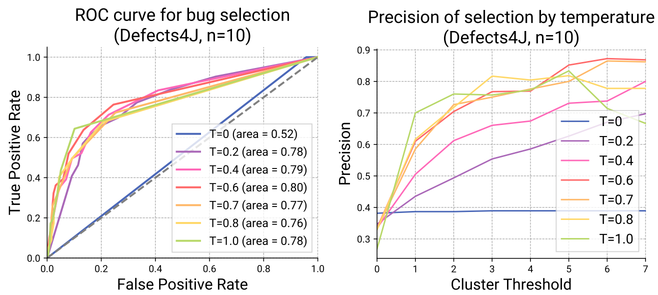
model	acc@1	acc@3	acc@5	prec@1	prec@3	prec@5
Codex	107	122	123	0.60	0.68	0.69
GPT-3.5-0613_p2	106	125	134	0.44	0.52	0.55
GPT-3.5-0301	92	112	115	0.49	0.60	0.61
StarCoder-15B	70	83	89	0.46	0.55	0.59
CodeGen2-16B	48	56	57	0.47	0.55	0.56
Bloom	45	54	56	0.45	0.54	0.56
Incoder-6B	29	33	34	0.38	0.43	0.45
GPT-3.5-0613_p1	18	22	23	0.31	0.37	0.39

TABLE 10: Ranking performance of Libro with different sampling temperatures (selection threshold $= 1$, $n = 10$)

Table 10 shows the performance of Libro after ranking FIBs among the selected bugs with a threshold of 1. Codex is still the best performing LLM after ranking, followed by the ChatGPT and StarCoder models. The performance of CodeGen2 and Bloom models are similar, and Incoder-6B shows the worst performance. The best performing open-source LLM, StarCoder, achieves about 60% precision when inspecting top five ranked tests. Overall, the relative performance of LLMs is preserved after additional postprocessing steps, and consistently improves the precision for all LLMs.

Answer to RQ5-1: The selection and ranking strategy of Libro is robust to the choice of LLM.

6.5.2 RQ5-2



(a)

(b)

Fig. 10: Selection performance by temperature.

As discussed earlier, the selection of temperature may have a significant effect on the selection and ranking algorithm of Libro, as it directly controls how repetitive multiple runs of LIBRO will be. Thus, we investigated how selection and ranking performance is affected by the choice of LLM sampling temperature. Results are shown in Figure 10. To our surprise, the selection performance of LIBRO, evaluated via the ROC-AUC metric that is independent to the selection of threshold, is remarkably consistent for different temperature values, with all showing an ROC-AUC of 0.76-0.80. We are unaware of prior work reporting similar results, and are similarly perplexed as to why ROC-AUC should stay so consistent under different temperature settings. It is clear that different temperature settings yield different behavior, in the expected way: looking at Figure 10(b), we see that for lower temperatures, cluster thresholds need to be larger to have high predictive power, as results are already very similar due to the low randomness of sampling; meanwhile, for higher temperatures, small thresholds may already be predictive of high precision as the outputs are diverse, and the fact that they overlap at all is significant. Nonetheless, these effects are counterbalanced and all temperatures end up leading to a similar bug selection performance. The one exception is T=0, as almost all results are identical for different runs in this scenario. (T=0 did not complete test output overlap in four cases; in one case, a test appeared to be flaky, while in three other cases, the tests would print timestamps, which was not abstracted during the clustering process.)

Temperature	acc@1	acc@3	acc@5	prec@1	prec@3	prec@5
T=0.0	58	58	58	0.38	0.38	0.38
T=0.2	78	91	93	0.29	0.34	0.35
T=0.4	88	111	115	0.26	0.32	0.33
T=0.6	99	122	125	0.26	0.32	0.33
T=0.7	97	117	123	0.26	0.31	0.33
T=0.8	94	115	120	0.26	0.31	0.33
T=1.0	60	85	86	0.19	0.26	0.27

TABLE 11: Ranking performance of LIBRO-StarCoder with different sampling temperature (selection threshold $= 1$, $n = 10$)

Ranking results by temperature are shown in Table 11. Similarly to the overall reproduction results presented in Figure 8b, sampling at T=0.6 shows the best performance, even considering both selection and ranking. Meanwhile, as apparent on the right side of Table 11, the precision of LIBRO decreases as the temperature increases, which may be related to the tendency for lower temperature outputs to generally yield better performance on a single run [11]. Within a moderate temperature range, the final ranking performance remains relatively stable. This suggests that LIBRO produces consistent results across multiple runs, enhancing the reliability and reproducibility of our approach.

> **Answer to RQ5-2:** Surprisingly, while results sampled from different temperatures have different characteristics, the performance of our selection algorithm is similar regardless of temperature. Considering all post-processing, T=0.6 showed the best performance, similarly to RQ4-4.

Listing 4: Generated FIB test for AssertJ-Core-2666.

```java
public void testIssue952() {
  Locale locale = new Locale("tr", "TR");
  Locale.setDefault(locale);
  assertThat("I").as("Checking␣in␣tr_TR␣locale").
       containsIgnoringCase("i");
}
```

Listing 5: Generated FIB test for Checkstyle-11365.

```java
public void testFinalClass() throws Exception {
    final DefaultConfiguration checkConfig =
            createModuleConfig(FinalClassCheck.class);
    final String[] expected = CommonUtil.EMPTY_STRING_ARRAY;
    verify(checkConfig, getPath("InputFinalClassAnonymousClass.java"
          ), expected);
}
```

7 DISCUSSION

7.1 Example output of LIBRO

Example outputs of LIBRO-Codex from the GHRB dataset are presented to provide further context for our quantitative results.

TABLE 12: Bug Report Successfully Reproduced: URLs are omitted for brevity (AssertJ-Core Issue #2666)

Title	**assertContainsIgnoringCase fails to compare i and I in tr_TR locale**
See org.assertj.core.internal.Strings#assertContainsIgnoringCase [url] I would suggest adding [url] verification to just ban toLowerCase(), toUpperCase() and other unsafe methods: #2664	

Table 12 shows our first example, a successful reproduction of issue #2685 from the AssertJ-Core project. The issue reports a bug for a particular locale (`tr_TR`), where the letter `I` is being mishandled. A successful test reproducing this issue from LIBRO is presented in Listing 4; it is noteworthy that LIBRO successfully generated this test even though the bug report does not include any executable code. Instead, it used the method names (`containsIgnoringCase`) provided within the bug report to generate a useful test. Furthermore, for this bug, a BRT is ranked at second place, meaning that a developer could quickly discover and use it.

TABLE 13: Bug Report Reproduction Failure: Lightly edited for clarity (Checkstyle Issue #11365)

Title	**FinalClassCheck: False positive with anonymous classes**

... I have executed the cli and showed it below, as cli describes the problem better than 1,000 words
`→src cat Test.java`
`[...]`
`public class Test {`
`  class a { // expected no violation`
`    private a(){} } }`
`[...]`
`→java [...] -c config.xml Test.java`
`Starting audit...`
`[ERROR] Test.java:3:5: Class a should be declared as final`
.	

Our second example, for which the bug report is presented in Table 13, is a case in which LIBRO failed to successfully reproduce the bug. Here, Checkstyle makes the incorrect inference that a particular class should be declared `final`, which leads to misleading results. One of the FIB tests generated by LIBRO for this bug report is presented in Listing 5. Because the Java file the test refers to on Line 5 does not exist within the project, the test fails for an inaccurate reason. Thus, this test makes a good cautionary example to illustrate the limitations of LIBRO, as it cannot change the execution environment of tests, which may be a prerequisite to reproduce a bug. Nonetheless, the test itself is functional; by inserting the Java code provided within the bug report into the appropriate file and re-running the test, the test is capable of reproducing the bug. Such results suggest that future work may also work on constructing the execution environment of tests to further widen the amount of bugs that can be automatically reduced, and thus alleviate developer effort on such tasks.

7.2 Code Overlap with Bug Report

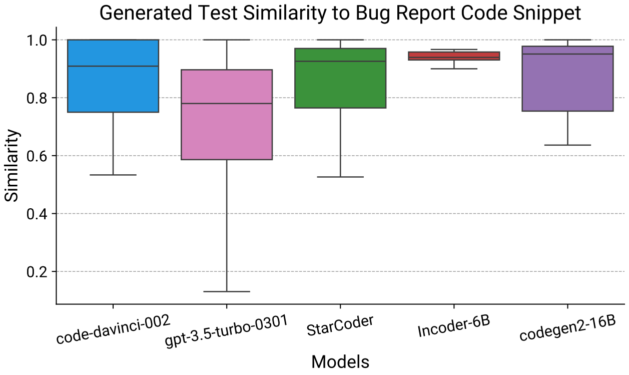


Fig. 11: Similarity between generated tests and code snippets contained in bug reports among multiple LLMs.

As Just et al. point out [2], bug reports can already contain partially or fully executable test code, but developers rarely adopt the provided tests as is. To investigate whether the performance of LIBRO is due to efficient extraction of report content or effective synthesis of test code, we analyzed the 750 bug reports from Defects4J used in our experiment, and compared them to tests generated by Codex (code-davinci-002). We find that 19.3% of them had full code snippets (i.e., code parsable to a class or method), while 39.2% had partial code snippets (i.e., not a complete class or method but in the form of source code statements or expressions); finally 41.5% did not contain code snippets inside. Considering only the 251 bug reports that LIBRO successfully reproduced, the portion of containing the full snippets got slightly higher (25.1%), whereas the portion of bug reports with partial snippets was 37.9%, and 37.1% did not have code snippets. When LIBRO generated tests from bug reports containing any code snippets, we find that on average 81% of the tokens in the body of the LIBRO-generated test methods overlapped with the tokens in the code snippets.

In Figure 11, we evaluate how similar the output of LLMs evaluated were to code snippets provided within the bug report. code-davinci-002, StarCoder, and CodeGen2 all showed a similar similarity between the generated test and snippets provided in the bug report, suggesting that they rely on the bug report to a similar degree; judging from the similarity distribution, these models could successfully reproduce bugs even when the exact reproducing snippet is not provided within the bug report. Meanwhile, Incoder-6B shows a very tight distribution around a similarity of 0.9, suggesting that most of its successes were due to very similar code already being provided within the bug report. Finally, gpt-3.5-turbo-0301 succeeded even when the similarity to the bug report was low; this may be due to its different training focus (natural language) and consequently even when the generated snippets did not resemble the provided code, it could succeed nonetheless.

Although full code snippets are often provided in bug reports, this does not necessarily mean that such code will successfully reproduce bugs; indeed, the Copy&Paste baseline succeeded far less often than LIBRO, as shown in Figure 2, only reproducing 36 bugs. Thus, we conclude that while LIBRO can be influenced by how much code is provided within a bug report, it nonetheless also succeeds in correcting non-reproducing code or even from generating test code from bug reports almost exclusively in natural language. In turn, this means that the LLMs used in our study can both extract the helpful parts of code snippets provided in bug reports, as well as synthesize tests from scratch, based on the given report.

7.3 Expected Impact for Developer Workflow

We envision that the results of this research could aid developers when processing bug reports. Given that the number of bug reports outstrips the amount of resources required to deal with and triage bug reports [4], and that reproducing a bug can be resource-intensive, we believe that LIBRO would reduce that effort and streamline developer workflow. When LIBRO succeeds in generating FIB tests, the tests generated by LIBRO could help triage based on code coverage, and also help debugging via a concrete reproducing test. Even when LIBRO fails to generate FIB tests, it could be an indication that the quality of the bug report is low, and thus potentially save developer inspection effort. While not yet commonly part of developer workflow, the tests generated by LIBRO could also be used as input for automated debugging algorithms and reduce a bottleneck in their usability, as noted in the introduction.

As with any technique, LIBRO is not perfect, and should be used with caution. As LLMs can generate incorrect, yet plausible, results, it is recommended to repeat the LLM queries multiple times and use the confidence measure defined in our paper to understand the likelihood of LLM correctness. Furthermore, while GPT models show strong performance, as our results show, their behavior can change drastically between different versions. To provide a consistent and privacy-conscious service, our results also suggest using LLMs from the StarCoder family of LLMs can be a good choice.

8 Related Work

8.1 Test Generation

Since at least the 1970s, automatic test generation has been a topic of research in software engineering [43]. In the long period since, there have been multiple shifts in approaches, such as the shift that occurred when object-oriented programming languages gained popularity, which necessitated the derivation of method call sequences [44], [45]. An often-cited problem with test generation techniques is the oracle problem [46], which stipulates the difficulty of automatically determining what the correct behavior of a software system should be. As a result of this difficulty, test generation techniques will rely on so-called implicit oracles such as crashes [44], or assume a regression testing scenario and treat the current output of the system as correct [45], [47]. Still others will use rule-based heuristics to derive oracle behavior from structured specification documents [48]; these techniques may suffer in performance when the structure changes, whereas LIBRO makes no assumptions about the structure of bug reports.

Meanwhile, there is a body of literature that focuses on reproducing bugs, similarly to LIBRO. The oracle problem is nonetheless an obstacle, so program crashes are often used as a proxy for reproducing a bug [46]. Most existing work on reproducing crashes focuses in particular on reproducing a crash stack trace [16], [49], [50], [25], [51]. On the other hand, Yakusu [52] and ReCDroid [5] analyze formatted bug reports to automatically reconstruct a sequence of actions that leads to crashes in mobile applications, and encapsulate those sequences in tests. All the aforementioned work differs from LIBRO, as they use the implicit crash oracle to discern whether a bug was reproduced and thus can only deal with crash bugs. Meanwhile, BEE [53] proposes a technique to parse bug reports and extract the observed or expected behavior from natural language descriptions, but does not generate tests. To the best of our knowledge, we are the first to propose a technique to reproduce general bug reports in Java projects.

Since our previous publication [10], significant research progress has been made in generating tests using LLMs. Lemieux et al. [17] combine LLM generation into an SBST loop to achieve better results than just using traditional SBST or LLMs alone. Liu et al. [54] generate text input for tests of mobile applications, and demonstrate that generating text input can help improve coverage. Closely related to our work, Feng et al. [55] propose AdbGPT, a technique that automatically reproduces bug reports for mobile applications. Our work is distinct from the aforementioned techniques, as (i) it targets the reproduction of bugs in general software, and (ii) we provide a comparison of the performance of a large number of LLMs in generating tests. In particular, we are unaware of such a comparison of LLMs being made, and thus believe our experiments may contribute to the software engineering community.

8.2 Code Synthesis

Similarly to test generation, code synthesis has been researched for a long time. The traditional approach to code synthesis was to use SMT solvers within the framework of Syntax-Guided Synthesis (SyGuS) [56]. Recent research on code synthesis has also used machine learning techniques, which also yield strong performance; for example, Chen et al. [7] demonstrated that LLMs could effectively synthesize Python code from a natural language description. To further improve the performance of code synthesis, some techniques have also generated tests: AlphaCode used automatically generated tests to boost their code synthesis performance [57], while CodeT jointly generated tests and code from a natural language description [58]. However, these techniques focus on code synthesis, not test generation, and thus the tests are discarded after evaluating the generated code. In contrast, LIBRO is fully focused on test generation, and furthermore introduces a novel pipeline to reduce developer effort in inspecting generated results.

9 Conclusion

In this paper, we introduce LIBRO, a technique that uses a pretrained LLM to analyze bug reports, generate prospective tests, and finally rank and suggest the generated solutions based on a number of simple statistics. Upon extensive analysis, we find that LIBRO using OpenAI's code-davinci-002 LLM is capable of reproducing a significant number of bugs in the Defects4J benchmark as well as generalize to a novel bug dataset that was not part of its training data; furthermore, we demonstrated that LIBRO could indicate when its tests were likely to actually reproduce the bug. Our additional large-scale experiments comparing the bug reproducing performance of 15 LLMs reveal that open-source LLMs can also show strong performance, with the StarCoder LLM showing the best performance among open-source LLMs evaluated, and other confirmations such that the size of the LLM positively influences bug reproduction performance. Our evaluation of our selection and ranking techniques also show that they are capturing general properties of LLMs for bug reproduction, as the heuristics work in the same manner over all LLMs evaluated. We hope that our experiments and results are of use to both researchers and practitioners when deciding which LLM would be appropriate for their application, and plan to continue researching the productive capabilities of open-source LLMs.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, ser. eclipse '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 35–39. [Online]. Available: https://doi.org/10.1145/1117696.1117704

[2] R. Just, C. Parnin, I. Drosos, and M. D. Ernst, "Comparing developer-provided to user-provided tests for fault localization and automated program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 287–297.

[3] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 572–583. [Online]. Available: https://doi.org/10.1145/3180155.3180175

[4] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "Ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 314–325. [Online]. Available: https://doi.org/10.1145/3338906.3338935

[5] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G.J. Halfond, "Recdroid: Automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 128–139.

[6] 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.

[7] 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.

[8] P. O'Hearn, "Formal reasoning and the hacker way." 2020, keynote for the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). [Online]. Available: https://www.youtube.com/watch?v=bb8BnqhY3Ss&ab_channel=ICSE

[9] 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*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[10] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, pp. 2312–2323. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00194

[11] 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," 2023.

[12] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, and C. M. et al., "Starcoder: may the source be with you!" 2023.

[13] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, pp. 61–70. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00015

[14] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[15] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 96–138, Jan 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09762-1

[16] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 101–110.

[17] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023.

[18] L. Chen, M. Zaharia, and J. Zou, "How is chatgpt's behavior changing over time?" 2023.

[19] "Openai gpt-3.5 model documentation." [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5

[20] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, and A. S. L. et al., "Bloom: A 176b-parameter open-access multilingual language model," 2023.

[21] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," 2022. [Online]. Available: https://arxiv.org/abs/2205.11916

[22] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[23] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 165–176. [Online]. Available: https://doi.org/10.1145/2931037.2931051

[24] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2228–2240. [Online]. Available: https://doi.org/10.1145/3510003.3510040

[25] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen, "Single-objective versus multi-objectivized optimization for evolutionary crash reproduction," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds. Cham: Springer International Publishing, 2018, pp. 325–340.

[26] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo, "The github recent bugs dataset for evaluating llm-based debugging applications," 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. Muennighoff, T. Wang, L. Sutawika, A. Roberts, S. Biderman, T. L. Scao, M. S. Bari, S. Shen, Z.-X. Yong, H. Schoelkopf, X. Tang, D. Radev, A. F. Aji, K. Almubarak, S. Albanie, Z. Alyafeai, A. Webson, E. Raff, and C. Raffel, "Crosslingual generalization through multitask finetuning," 2023.

[29] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2023.

[30] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," 2023.

[31] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig, "Pal: Program-aided language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 10 764–10 799.

[32] C. Thunes, "javalang: Pure Python Java parser and tools," https://github.com/c2nes/javalang, 2022.

[33] R. Premraj, T. Zimmermann, S. Kim, and N. Bettenburg, "Extracting structural information from bug reports," in *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*, ACM Press. New York, New York, USA: ACM Press, 05/2008 2008, pp. 27–30.

[34] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," 2020.

[35] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Martin, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, pp. 1936–1964, 2016.

[36] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," 2023.

[37] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[38] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[39] OpenAI, "Gpt-4 technical report," 2023.

[40] ——, "Reproducible outputs with openai's text generation api," 2023, accessed: 2024-07-01. [Online]. Available: https://platform.openai.com/docs/guides/text-generation/reproducible-outputs

[41] A. Narayanan and S. Kapoor, "Is gpt-4 getting worse over time?" 2023. [Online]. Available: https://www.aisnakeoil.com/p/is-gpt-4-getting-worse-over-time

[42] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022.

[43] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[44] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and*

applications companion. New York, NY, USA: ACM, 2007, pp. 815–816.

[45] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[46] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[47] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *CoRR*, vol. abs/2009.05617, 2020. [Online]. Available: https://arxiv.org/abs/2009.05617

[48] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 188–199.

[49] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE transactions on software engineering*, vol. 41, no. 2, pp. 198–220, 2014.

[50] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 910–913.

[51] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for java," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1278–1282.

[52] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 141–152. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213869

[53] Y. Song and O. Chaparro, "Bee: A tool for structuring and analyzing bug reports," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1551–1555. [Online]. Available: https://doi.org/10.1145/3368089.3417928

[54] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1355–1367. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00119

[55] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," 2023.

[56] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, M. Irlbeck, D. A. Peled, and A. Pretschner, Eds. IOS Press, 2015, vol. 40, pp. 1–25. [Online]. Available: https://doi.org/10.3233/978-1-61499-495-4-1

[57] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with AlphaCode," *arXiv preprint arXiv:2203.07814*, 2022.

[58] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.