Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Effective Prompting Strategies for Automated Test Case Generation in Bug Reproduction

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Martin Kunz
geboren am: 15.10.1996
geboren in: Hadamar

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Yannic Noller

eingereicht am: ..... verteidigt am: .....

Abstract

Test case generation is an important part of modern software development, especially for reproducing reported bugs and avoiding future regressions. While conventional approaches for automated test generation often focus on structural metrics such as code coverage, they often lack the ability to generate semantically precise tests from natural language bug reports. The current rapid development of large language models is creating new and promising opportunities here. Large language models can synthesize tests directly from bug reports through targeted prompting. The recently published LIBRO approach uses large language models to generate test cases for bug reproduction by converting bug reports into structured prompts and passing them to the model multiple times [1]. The generated tests are then post-processed, embedded in existing test classes, automatically executed and ranked according to heuristics that predict successful reproductions with a high probability.

In this paper, we extend the LIBRO approach with the aim of systematically analyzing the influence of individual prompt components. To this end, we first derive a default prompt based on common features of successful prompts from LIBRO and related work. This prompt is then specifically varied in an experimental setup: Individual components (so-called prompt ingredients) are removed or added in order to evaluate their influence on the quality and success rate of the generated tests. The aim is to gain a better understanding of which elements are particularly relevant for successful bug reproduction by large language models. The evaluation is based on the data sets Defects4J (version 2.0) [2] and GHRB (GitHub Recent Bugs) [3], which are also used in the LIBRO paper.

Our results show that GPT-4o mini partially reproduces the behavior of LIBRO, but achieves significantly lower absolute success rates. The ablation study also confirms that Code Prefixes, Javadocs, and Few-Shot Examples are particularly crucial for successful bug reproduction.

Contents

1 Introduction

Software applications are increasingly being used in safety-critical areas where reliability is of the utmost importance. Systems in medical technology, banking, or the automotive industry, such as autonomous vehicles, must meet extremely high reliability and safety requirements, as malfunctions can have serious consequences. To minimize such risks, extensive testing is necessary to cover as many usage scenarios and potential edge cases as possible. However, with the growing complexity of modern software systems, this task is becoming increasingly difficult and time-consuming, which can jeopardize the reliability of such safety-critical systems. Although automated testing methods such as fuzzing have made good progress in some areas, they often reach their limits. They detect malfunctions, but rarely explain why a program fails or under what conditions the error occurs. For developers, however, this is essential, as error reproduction is usually the first step in fixing a bug.
The manual creation of such bug reproduction tests is time-consuming and error-prone. Here, recent advances in artificial intelligence, especially in large language models, offer new perspectives and possibilities. Large language models can process natural language bug descriptions and convert them into executable code. A prominent example is LIBRO [1], which uses large language models to automatically generate test cases from bug reports. LIBRO demonstrated for the first time that large language models can successfully reproduce about one-third of the bugs in the Defects4J benchmark [1].

In this thesis, we build upon the work of Kang et al. [1] and extend the LIBRO framework. As illustrated in Figure 1, LIBRO consists of four main stages: (A) Prompt Engineering, where a structured prompt is constructed from the bug report and optionally augmented with example reports and tests, (B) LLM Querying, where the model is prompted multiple times to generate candidate test cases, (C) Post-processing, which integrates the generated tests into the target project's test suite and resolves dependencies, and (D) Selection & Ranking, where candidates are executed, filtered, and ranked according to their likelihood of reproducing the bug. A detailed description of each stage of the LIBRO framework, including its implementation details and heuristics, is provided in Section 3.
Our contribution focuses specifically on stage (A): prompt engineering. While LIBRO already explored several prompt variants (e.g. with or without examples, stack traces, or constructor information), the space of possible prompt configurations remains large, and the effect of each component has not been studied in isolation. In this work, we systematically ablate and recombine the individual prompt components to quantify their contribution to successful bug reproduction. Furthermore, we investigate prompt augmentation by introducing new information sources, such as automatically extracted class or API documentation, to provide the large language model with richer context. By refining this crucial first step in LIBRO's pipeline, we aim to improve the overall effectiveness and reliability of large language model-based bug reproduction.

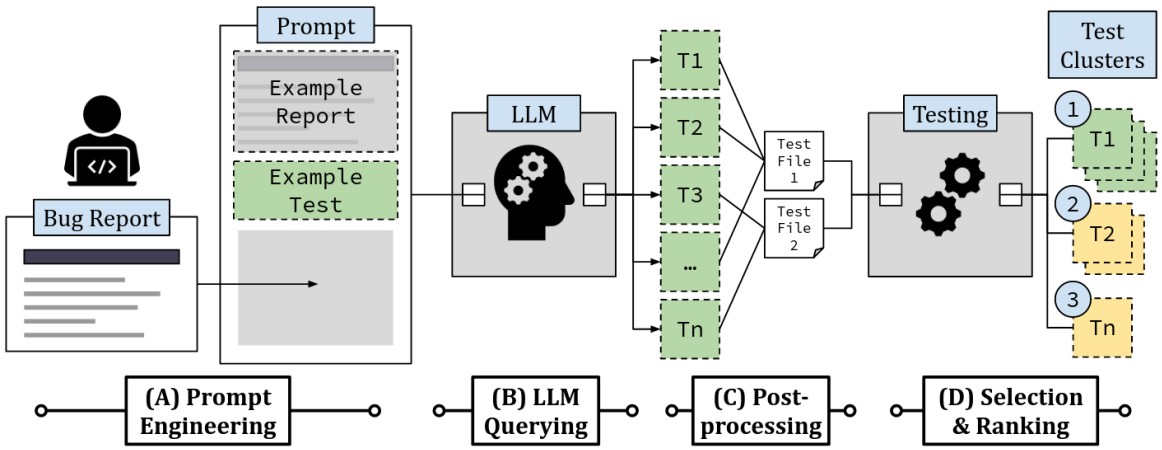


Figure 1: Overview of the LIBRO framework [1].

The aim of this master's thesis is to systematically evaluate and expand the LIBRO framework. First, we examine the extent to which the results presented in the original paper can be reproduced, particularly when using a large language model such as GPT-4o mini that differs from the one used in the original paper. Building on this, we conduct an ablation study to investigate the influence of individual prompt components and expand the framework with additional contextual information. The evaluation is guided by the following research questions:

RQ1 To what extent can the results presented in the LIBRO paper be reproduced?

RQ2 Which prompt components are particularly important for successful bug reproduction?

RQ3 Does adding additional information to the existing prompt components lead to improved results?

In summary, this thesis makes the following contributions:

- Conducting a comprehensive reproducibility study of the results reported by Kang et al. [1] for the LIBRO framework, using GPT-4o mini instead of Codex to evaluate the robustness and reproducibility of large language model-based bug reproduction.
- Conducting a fine-grained ablation study of the prompt components of LIBRO (e.g. bug description, examples, stack traces) to identify which elements contribute most to successful bug reproduction.

- Design and evaluation of new prompt components, such as automatically extracted class and API documentation, to investigate whether extended contextual information improves the reproduction rate based on the metrics from the original paper to ensure comparability.

The remainder of this thesis is structured as follows: Section 2 reviews the state of the art in automated test generation, large language model-based bug reproduction, and prompt engineering. Section 3 introduces the theoretical foundations, explains the LIBRO framework, and describes the benchmarks used in this work. Section 4 presents the importance of reproducibility and provides an overview of known problems and challenges. Section 5 details the experimental setup and evaluation metrics. Section 6 outlines the proposed approach for prompt ablation and prompt augmentation. The results of the experiments are reported in Section 7. Section 8 discusses the findings and also discussing threats to validity. Section 9 highlights possible directions for future research, summarizes the key contributions of this thesis and provides concluding remarks.

2 Related Work

In this Section, we present the related work, literature, and current research directions.

Prompt-based Approaches One of the first approaches to bug reproduction using large language models was presented by Kang et al. They present LIBRO, which uses Codex to generate candidates for reproduction tests from bug reports. To implement this, the bug report is converted into a structured Markdown format with an error description, a request to reproduce the error, and a code block as the start of a JUnit test. This prompts the model to write a test case that reproduces the bug. However, the model is not able to execute the code itself. The model's output is available as a txt file and is completed by a post-processing step and executed for both the buggy and fixed versions. Based on these results, the tests are ranked to identify valid bug reproductions. LIBRO achieved good results on the Defects4J dataset. For 33% of the bugs, at least one failed test could be generated. This corresponds to 251 of the 750 bugs within the Defects4J dataset. However, further testing of data that was not part of the Codex training also produced similar rates of 32%. Nevertheless, there are clear limitations here, as almost 67% of all bugs remained unreproduced and the accuracy was highly dependent on heuristics for evaluating the results [1]. Several further developments and studies based on LIBRO have already been published. A feasibility study of GPT-based models by Plein et al. showed that ChatGPT generated an executable test case for around 50% of the Defects4J bug reports, but only achieved fail-to-pass behavior in 6% of cases, i.e. a failure of the test before the fix and a success after the fix [4].

Agent-based Approaches Mündler et al. [5] transferred the concept of LIBRO to Python, the SWT-Bench dataset, and achieved a bug reproduction rate of about 15%. The SWE-Agent for automated bug fixing presented by Yang et al. [6] performed well in this context. Although actually developed for automated program repair, SWE Agent generated a suitable bug reproduction test in 15.9% of cases, outperforming other non-agent-based methods [7]. The further development, also by Mündler et al. [5], SWE-Agent+, which was explicitly instructed to execute and evaluate the generated tests, was able to further improve the previous result with 18.5% [5]. The results show that agent-based large language model systems perform better than models with one-time prompt queries.
Other recent work provides further promising results. Cheng et al. [8] present the BRT Agent, a large language model-based approach that has been applied on a larger scale in cooperation with Google. The approach uses a fine-tuned code editor large language model as an agent. For this work, the LIBRO idea was combined with tools from Google's development environments. The tool achieved a bug reproduction rate of 28% among 80 real and manual bug reports from the Google issue tracker. This is a better result than the 10% achieved by LIBRO. Furthermore, the authors transferred the generated tests to Google's internal Automated Program Repair (APR) system and were able to increase the rate of automatically fixed bugs by 30%.

At the same time, Otter was introduced, a system that combines prompting with a self-reflective planning module and rule-based analyses [9]. Otter takes an issue report and code base as input and dynamically decides which parts of the code to read and which tests to write. With GPT-4 as a large language model, Otter achieves a fail-to-pass rate of 31.4% on the TDD-Bench-Verified dataset. With Otter++, the further development of Otter, which executes test generation five times with heterogeneous prompts, the results were increased to 37% compared to Otter [9]. Otter and Otter++ thus deliver promising results for automatic test generation from code issues. Nevertheless, there is still room for improvement in these approaches.

Evaluation Challenges and Prompt Sensitivity In addition to developing new tools for automated bug reproduction, current research is also increasingly focusing on the limitations and evaluation problems of LLM-based test generation. Several studies show that tests generated by LLMs are often syntactically correct or compilable, but cannot necessarily correctly map the semantic conditions of a bug. In the context of bug reproduction in particular, this means that generated tests are executable but do not exhibit consistent fail-to-pass behavior.

Qureshi and Jiang [10] systematically investigate this problem by proposing a cognitively layered evaluation framework based on Bloom's taxonomy. Their study shows that LLMs deliver solid results on simple tasks such as code completion or syntactic test generation, but perform significantly worse when a deeper understanding of the cause of the error, the program logic, or the implicit assumptions in the bug report is required. The authors conclude that the lack of explicit semantic cues in the prompt is one of the main causes of incorrect or incomplete test cases [10].

A similar picture emerges in research on LLM-based automated program repair (APR), where reproducible tests often serve as a prerequisite for successful repairs. A recent overview by Yang et al. [11] analyzes over 30 APR approaches and shows that the performance of LLM-based systems strongly depends on the quality of the input contexts and prompt structures. In particular, work that integrates additional contextual information such as documentation, code snippets, or structured instructions achieves significantly better results than pure One-Shot prompt approaches [11].

These observations are further supported by an empirical study by Sun et al. [12]. The authors show that even small variations in the prompt structure, such as adding or removing individual instructions, lead to significant differences in the quality of the generated patches. The study emphasizes that prompt engineering should not be considered a purely heuristic step, but rather an independent influencing factor with a measurable effect on the behavior of LLMs [12].

Summary and Research Gaps Recent research in the field of automated test case generation using large language models has made considerable progress in the recent past. LIBRO and its successors show that large language models are capable of generating executable tests based on bug reports in natural language. Nevertheless, much of this work focuses primarily on developing new frameworks and increasing quantitative

success rates, while questions about the reproducibility of the respective approaches and explainability are not addressed.

This master's thesis addresses two central research gaps:

(1) Reproducibility of Existing Results: We systematically investigate the extent to which the results reported in the LIBRO paper can be replicated and reproduced with today's models. In doing so, we consider aspects that go beyond the original publication, such as replacing the outdated Codex model with modern large language models.

(2) Systematic Analysis of Prompt Components: Instead of adopting the original LIBRO prompt unchanged, this work examines the influence of individual prompt elements on the quality of the generated tests. While Kang et al. [1] have already evaluated various prompt configurations, there have been no fine-grained investigation of the individual components, the addition of documentation, and their combinatorial effect.

By combining both perspectives, this work contributes to the explanation, robustness, and traceability of large language model-based test generators. The goal is to better understand why a bug reproduction test is generated by the model and how targeted prompt engineering can improve the reliability and stability of such systems. In contrast to previous studies, the focus is therefore not on maximizing metrics, but on the transparency and optimization of existing methods. The work thus closes a relevant gap and provides practical insights for the further development of reproducible and explainable large language model-based test case generation.

3 Background

In this work, we build on the concept of large language model-based error reproduction to increase the efficiency and precision of automatic test generation. Instead of focusing on post-processing steps, we concentrate on iteratively refining prompts by adding or removing specific components. In addition, the Javadoc of the corresponding methods and classes of the projects is added as an additional prompt component to improve the large language model's understanding of the expected program semantics.
In this Section, we define the necessary terminology and present the required fundamentals of our approach. This includes an introduction to large language models, an overview of test generation from bug reports, and a detailed presentation of the framework on which we build.

3.1 Terminology

The following subsection defines the key terms that are essential for understanding and implementing the approaches presented in this paper.

Definition 1 (Large Language Model (LLM)). Large Language Models have emerged as artificial intelligence systems that can process texts, generate coherent communication, and generalize to multiple tasks, such as coding tasks [13].

Definition 2 (Prompt). A prompt is defined as the instruction used to specifically control and guide the output of a large language model. In terms of its structure, a prompt can take two primary forms: either as a natural language instruction that provides the model with the necessary context for guidance, or as a learned vector representation that activates relevant knowledge. Regardless of its form, the prompt typically consists of three crucial elements that significantly shape the LLM's response: the instruction, the context, and the user input [14].

Definition 3 (Prompt Engineering). Prompt engineering is a technique for extending the capabilities of large language models. This method involves the systematic design of prompts to control the model output. The overall goal of prompt engineering is to increase and optimize model efficiency and elicit the desired model behavior. A key feature of this approach is that it is achieved without modifying the core model parameters. Instead of updating the parameters, prompt engineering enables the seamless integration of pre-trained models into downstream tasks [14].

Definition 4 (Few-Shot Learning). Few-Shot Learning in LLMs refers to a model's ability to solve new tasks based solely on a few example demonstrations that are passed to it in the prompt during inference time without gradient updates or fine-tuning the model. Input-output pairs are used as conditions to specify the desired task, allowing the model to derive the underlying pattern from these examples and apply it to new instances [15].

Definition 5 (Bug Report). A bug report is a structured message that allows users of a software system to report an observed problem or malfunction to developers. It serves as a means of communication between users and developers and contains information necessary to identify, reproduce, and fix the bug, such as the observed malfunction, the expected behavior, technical environment data, or reproduction steps. The quality and completeness of this information have a significant impact on how effectively developers can reproduce and fix the bug, and at the same time form a central basis for the automated analysis and processing of error messages [16].

Definition 6 (Bug Reproducing Test (BRT)). A bug reproducing test is defined as a test that fails precisely when the bug specified in the report is present. For evaluation purposes, a test is treated as a BRT if it fails on the version that contains the bug specified in the report and, in contrast, passes on the version that fixes the bug [1].

Definition 7 (Fail in Buggy program (FIB)). A test is referred to as Fail In the Buggy program if it meets the necessary condition to be a BRT: it must compile and fail in the buggy program. FIB tests are identified by running the test in the buggy target program and verifying that it has no compilation errors and fails [1].

Definition 8 (Javadoc). Javadoc is defined as the JDK tool used to generate API documentation. This tool processes special comments in Java source code, which are referred to as documentation comments. The task of Javadoc task is to define the official Java Platform API Specification. This specification represents a contract between callers and implementations and must be implementation-independent. The documentation should describe all aspects of the behavior of each method that a user can rely on, with the goal of providing all necessary assertions for a "write once, run anywhere" implementation of the Java platform [17].

3.2 Large Language Models

This Section introduces key concepts that determine the behavior and text generation of LLMs. The aim is to provide a basic understanding of how LLMs calculate probabilities for possible outputs and how these probabilities can be controlled by different sampling strategies.
First, we explain how the raw values that the model calculates for each possible token are generated and how the probabilities for the next token are derived from them. Next, we discuss the temperature parameter, its influence on the model's output, and its effects on the creativity or determinism of the LLM's generation. Finally, the last subsection explains how limiting the model to the most probable tokens can be used to control which outputs the model prefers to select.
These three concepts form the basis for specifically influencing the behavior of LLMs in text or code generation and understanding the balance between determinism, diversity, and quality of the generated content.

3.2.1 Logits

To understand how generation parameters such as temperature influence the behavior of LLMs, it is necessary to first examine the underlying token selection mechanism. In particular, the concept of logits and their transformation into probabilities form the mathematical basis for all subsequent sampling strategies.
Logits form the central link between the internal representation of an LLM and the actual output generated. They represent the raw outputs of the last linear layer of the model before an activation function, such as the softmax function, is applied. Each logit corresponds to a numerical value that describes the model's relative preference for a particular token. The higher this value, the more likely this token will be selected in the next generation step.
In the context of an autoregressive language model that generates tokens one by one, input sequences are first converted into vectors that are processed by the model. This creates an internal representation for each token, known as the hidden state h. This vector contains the semantic and syntactic information captured in the context so far. To derive a prediction about the next token from this representation, the hidden state is transformed into the output space by a linear projection. The resulting logits are described by the following equation:

$$logits = h \cdot W^T + b \tag{1}$$

Here, h stands for the hidden state of the model, i.e. the contextual representation of the previous input. W denotes the weight matrix of the output layer, which determines the mapping between the internal representation and the vocabulary space. b stands for the bias vector, which allows a shift in the values, and W^T describes the transposed weight matrix, which is necessary for correct dimension adjustment in matrix multiplication [18]. The result of this calculation is a vector that contains a value for every possible token in the vocabulary. However, these values are not yet probabilities, but rather unnormalized scalar values whose scale and ratio are decisive for further sampling. Only by applying the softmax function are these converted into probabilities:

$$P(x_i) = \frac{e^{\text{logit}_i}}{\sum_j e^{\text{logit}_j}} \tag{2}$$

This calculation ensures that all values are between 0 and 1 and that the sum of all probabilities equals 1. The model then selects the next token based on this distribution. This is done by selecting the token with the highest probability [18].
Logits thus play a crucial role in the generation process of LLMs, as they directly determine how likely a particular token is to be output next. Changes in the logits, such as through scaling, normalization, or sampling strategies, have a direct impact on the output of the LLM and thus on the style, creativity, and consistency of the output. They are therefore an important factor in controlling the behavior of LLMs and can be used specifically for tasks such as code generation, error reproduction, or semantically

consistent text output [18].

3.2.2 Temperature

After the logits $u_{1:|V|}$ have been calculated, they are transformed into a probability distribution over the vocabulary V using the softmax function. Temperature scaling is often used to further control the output diversity. This scales the logits before applying the softmax function [19]. This yields:

$$p(x = V_l \mid x_{1:i-1}) = \frac{\exp(u_l/t)}{\sum_{l'} \exp(u_l'/t)} \tag{3}$$

where $t \in (0, \infty)$ represents the temperature. The temperature parameter influences the entropy of the resulting distribution and thus directly influences the diversity of the generated outputs [20].

For **small temperature values** $0 < t < 1$, the logits are increased by division, which amplifies the differences between the probabilities. This makes the resulting distribution sharper, and the model is highly likely to select the token with the largest logit. In the limiting case $t \to 0^+$, the distribution approaches deterministic *argmax* sampling, in which only the most probable token is ever selected. This can be useful when consistent and reproducible outputs are required, but it severely limits variability [21].

In addition, for **higher temperature values** $t > 1$, however, the logits are weakened, resulting in a smoother distribution in which tokens with originally lower probabilities are also selected more frequently. This increases the diversity and creativity of the generated outputs, but also the risk of incoherent or erroneous tokens [21].

Therefore, the choice of temperature directly influences the trade-off between determinism and diversity. Low temperatures lead to more stable but less diverse results, while high temperatures produce more exploratory but potentially less reliable outputs. In the context of LLM-based test case generation, as used in LIBRO, this balance is particularly important: too low a temperature can lead to repetitive, redundant tests, while too high a temperature can generate many erroneous or uncompilable test cases. This thesis chooses a moderate temperature ($t = 0.7$), as also proposed in the original LIBRO paper, to ensure a good balance between reproducibility and diversity of the generated tests [1].

3.2.3 Top-k Sampling

Top-k sampling is a common strategy for controlling text generation in LLMs. It is a heuristic sampling method that aims to control the balance between text quality and diversity by pruning the probability distribution over possible next tokens.
While a language model generates a probability distribution $p = (p_1, p_2, \ldots, p_{|V|})$ across all tokens in the vocabulary V after applying the softmax function, top-k sampling limits

the generation to the k most probable tokens. The idea behind this approach is to completely discard unlikely tokens with very low probability in order to avoid incoherent or illogical outputs.
Formally, the transformation of the original probability distribution according to Nadeem et al. is defined as follows [22]:

$$\hat{p}_i = \frac{p_i \cdot \mathbb{1}_{\{i \leq K\}}}{\sum_{j=1}^{K} p_j} \tag{4}$$

Here, p_i denotes the original probability of token i, i.e. the value calculated by the model with which this token could be selected next. The indicator function $\mathbb{1}_{\{i \leq K\}}$ only takes the value 1 for the K most probable tokens, thus ensuring that only these tokens are considered in the further sampling process. All other probabilities are set to 0. The resulting, newly normalized probability $\hat{p}_i$ ultimately describes the distribution within this reduced set of the top 1 tokens, with the sum of all remaining probabilities again equaling 1.

All tokens outside this set are set to 0. They are therefore no longer taken into account. This truncation reduces the entropy of the distribution $H(\hat{p}) < H(p)$, resulting in more focused and less random text generation.
The parameter K is crucial for the behavior of the model:

- **Smaller values** of K (e.g. 10–50) result in coherent, deterministic, and high-quality texts, but with less diversity.
- **Larger values** of K (e.g. 100-1000) increase diversity, but can lead to semantic errors, as tokens with lower probabilities are also taken into account.

Top-k sampling follows certain mathematical properties that are crucial for high-quality text generation. These include:

1. **Entropy Reduction:** The entropy of the distribution is reduced, eliminating random tokens.
2. **Order Preservation:** The ranking of token probabilities is preserved. The most probable tokens remain dominant.
3. **Slope Preservation:** The relative ratio between the probabilities of the remaining tokens is preserved. This ensures semantic consistency.

These properties ensure that the sampling process generates consistent and plausible text sequences without completely losing diversity. In combination with other techniques such as temperature scaling, top-k sampling can be fine-tuned to achieve the desired degree of creativity or precision.

3.3 Test Generation for Bug Reproduction

The automated generation of test cases is a central discipline of software quality assurance. While classic test generation methods primarily target structural coverage, robustness

testing, or random inputs, test generation for bug reproduction pursues a more specific goal: the targeted reproduction of a known error. Such a test case not only serves to validate a fix, but also to ensure that the same bug does not occur again in the future. The development of automated methods for bug reproduction has advanced considerably in recent years, from heuristic, search-based, and symbolic approaches to newer methods based on large language models. The following Subsections first present classic methods for test case generation, before describing modern, LLM-based approaches that enable semantic and contextual analysis.

3.3.1 Conventional Methods of Test Case Generation

Before artificial intelligence, specifically LLMs, found its way into software testing research, automated test case generation was mostly based on structural and data-driven methods. The goal was to generate inputs or test sequences that cover a specific program behavior as completely as possible or reveal potential error states [23].

A fundamental approach is **Random Testing**, in which random inputs are generated and executed in the program to provoke crashes or unexpected outputs [24]. This method is easy to implement, but suffers from low efficiency, as randomly generated inputs often do not cover the relevant code paths. Fuzzing is used to control the random selection in a more targeted manner. Mutation-based and generation-based fuzzing are particularly relevant. While mutation fuzzers slightly modify existing inputs, generative fuzzers use syntactic knowledge about the input format (e.g. grammar rules) to generate valid but varied inputs. These fuzzers are used in many security-critical applications, for example to detect memory and input processing errors in system libraries or parsers [25,26].

Another important class is formed by **Search-Based Software Testing (SBST)** approaches, which generate test data using evolutionary algorithms or other optimization methods [27]. Here, an objective function is defined, such as maximum code coverage or the targeted triggering of certain exceptions. Genetic operators such as mutation and selection are used to iteratively improve inputs until a specific criterion is met. These methods are particularly suitable for reaching and testing program states and code sections that are difficult to access.

In addition, there are **Symbolic and Constraint-based** methods that derive program inputs systematically using logical conditions rather than randomly. Tools such as KLEE [28] or SAGE [29] symbolically analyze the possible execution paths of a program, generate constraints on input variables, and solve them using constraint solvers to find specific test data that cover certain paths. These techniques enable high precision, but are often limited by scalability issues and the complexity of real software.

These conventional methods are only of limited use for bug reproduction, i.e. the targeted recreation of an error known from a bug report. They usually require in-depth domain knowledge or manual preparatory work, such as extracting relevant inputs, creat-

ing special grammars or configurations from reports. Since bug reports are often written in natural language and lack contextual information, conventional techniques are not well suited to processing the semantic content of such reports. The use of LLMs opens up new possibilities for converting natural language error messages directly into executable tests.

3.3.2 LLM-based Approaches

LLMs have recently established themselves as a promising basis for automated test case generation, especially for tasks that require semantic understanding of natural language and source code. In contrast to the previously mentioned conventional approaches based on structural analysis or symbolic execution, LLM-based methods formulate test generation as a text-to-code problem: The model receives a natural language error description based on a bug report or a code snippet as input and generates an executable test case from it [1].
A key advantage of LLMs lies in their ability to capture contextual and semantic relationships between natural language and program behavior. Models designed specifically for coding tasks, such as Codex, can synthesize unit tests, correctly combine API calls, and even implicitly consider program states without the need for explicit static analysis [8, 9]. This makes them particularly suitable for bug reproduction, where error descriptions are often incomplete or ambiguous.
However, the success of such models depends heavily on prompt design and context provision. Research shows that carefully structured prompts, such as those that embed code context, sample tests, or precise instructions, significantly increase the success rate of bug reproduction, while minimalist prompts often only provide syntactically correct but semantically useless tests [1, 8].
Despite these advances, challenges remain in terms of reproducibility, interpretability, and generalizability. LLMs tend to deliver less accurate results or perform incorrect error localization outside their training context [9].
In the field of automated bug reproduction in particular, it is striking that systematic investigations into the effect of individual prompt components have been largely lacking to date. Many existing works use prompt templates without analyzing which components actually contribute to success. In addition, the reproducibility of the reported results is often not guaranteed precisely because of the use of LLMs, since LLMs are not deterministic in the classical sense. Although greedy decoding ($temperature = 0$, $top\text{-}k = 1$) generally expects deterministic token selection, recent studies show that differences still occur in real LLM inference environments [30, 31].

This work addresses this issue: Through a reproducibility study of the LIBRO framework and a systematic variation of the prompt components, we aim to investigate how different prompt structures affect the quality of the generated tests. The goal is to gain a better understanding of which elements are crucial for successful bug reproduction and how targeted prompting can improve the reliability and stability of LLM-based test generators. In this way, the work contributes to the transparency, traceability, and optimization of future approaches in the field of automated test case generation.

3.4 LIBRO: LLM-Induced Bug Reproduction

In software development, the automated generation of test cases is an important issue for guaranteeing and ensuring the quality of software systems. While traditional techniques such as Randoop or Whole Test Suite Generation often aim to maximize code coverage, the challenge remains of generating tests that reproduce specific semantic problems described in bug reports. Such reports are essential for developers, but converting them into executable test cases requires manual effort and a great deal of domain knowledge. As a study by the authors shows, tests created in response to bug reports account for a significant portion of the entire test suite. This is where the LIBRO framework presented by Kang et al. [1] comes in. It uses the capabilities of LLMs to automate this process. The core idea is to leverage the ability of LLMs to generate code-related tasks. However, since LLMs are not capable of executing the code themselves, LIBRO focuses on the downstream steps that enable the evaluation, validation, and prioritization of the test cases generated by the LLMs.
The paper presents LIBRO as a holistic approach that takes a bug report as input and delivers one or more potentially bug-reproducing test cases as output, accompanied by an assessment of their reliability. The entire process is divided into four main stages: Prompt Engineering, LLM Querying, Post-processing, and Selection & Ranking (see Figure 1).

The first step in the pipeline is **(A) Prompt Engineering**. Since LLMs function as autocomplete networks, the way a task is formulated significantly influences the quality of the output. LIBRO constructs a specific Markdown document from a given bug report, which serves as a prompt. The prompt format is structured in such a way that it prompts the LLM to write a test method. The prompt includes the following fixed components:

- The title of the bug report.
- A description of the problem.
- The command: `Provide a self-contained example that reproduces this issue.`
- The introduction to a code block with ```.
- The code prefix: `public void test`.

This structure explicitly prompts the LLM to create a test method that reproduces the described error. The authors experiment with different variations of this basic prompt to optimize performance. They found that adding example pairs (bug report and associated test case) from other projects improves performance, as LLMs benefit from such few-shot learning examples. In contrast, adding examples from the same project worsen performance, as the LLM tend to simply copy the examples. In addition, it was found that integrating stack traces in crashes improve reproduction for this type of error.

After creating the prompt, the next step in this pipeline is **(B) LLM Querying**. Here, LIBRO queries the LLM to generate a set of potential test candidates. LIBRO uses a non-greedy approach (weighted random sampling) by setting the temperature parameter to 0.7. This allows the LLM to generate several different test cases from the same prompt, as the selection of the next tokens is not strictly based on the highest probability. Generating multiple candidates increases the probability of finding a genuine BRT (Bug Reproducing Test).
The LLM output is limited so that only the actual test method body is accepted. At this point, the resulting code snippets are not yet compilable on their own, as they lack the necessary import statements and the correct class context.

The third step is **(C) Test Postprocessing**. Since the previously generated raw LLM outputs cannot be executed immediately, LIBRO integrates them into the existing test suite of the target project. This step consists of two main parts:

1. Insertion into a suitable test class: LIBRO finds the most suitable test class to insert the generated test method. The heuristic is based on the lexical similarity between the generated test method and the existing test classes. The match is calculated by the number of common tokens between the generated test and the test class. This approach mimics a developer's workflow and resolves many initial dependency issues.

2. Resolving remaining dependencies: After the insertion into a test class, there may still be unresolved dependencies. LIBRO attempts to resolve these heuristically. It parses the generated method to identify required classes, types, and constructors that have not been imported yet. LIBRO then searches for public classes with the identified name in the project classpath. If only one match is found, the corresponding import statement is added. In case of ambiguities, LIBRO chooses the most frequently used import statement for the respective class name throughout the project. Although this heuristic pipeline does not guarantee complete compilation, it is able to handle most unresolved dependencies.

The final step is the **(D) Selection & Ranking** of the generated tests. A bug reproducing test must fail in the buggy program and succeed in the corrected program. A necessary but not sufficient condition is that the test fails in the buggy program. Such tests are referred to as FIB tests (Fail In the Buggy program).
The framework attempts to decide when it makes sense to submit suggestions to a developer and which test should be given priority. This process is divided into two phases:

1. Selection: LIBRO decides whether results should be presented at all. The generated FIB tests are clustered according to their error output behavior (error type and error message). The core idea is that a high degree of consistency between several independent generations of the LLM indicates higher reliability. LIBRO only displays results if the size of the largest output cluster (`max_output_clus_size`) exceeds a certain threshold (`Thr`).

2. Ranking: Once the tests have been selected, they are sorted according to three heuristics:
    - Consistency with the bug report (`br_output_match`): Tests are preferred if their error message or test code directly contains behaviors or exceptions mentioned in the bug report.
    - Consensus (`output_clus_size`): Tests from larger output clusters are ranked higher, as this indicates a stronger consensus among the LLM generations.
    - Test length (`tok_cnts`): Shorter tests are preferred because they are easier for developers to understand and review.

First, syntactically identical tests are removed. Then, the output clusters themselves are sorted according to the heuristics, and within each cluster, the tests are also sorted. LIBRO presents the tests from different clusters alternately to offer a variety of solutions. This approach increases the probability that a genuine BRT will be among the first suggestions. Therefore, this strategy has the potential to significantly reduce the manual review effort for developers.

4 Reproducibility

This Section discusses the importance of reproducibility in empirical software engineering and machine learning research. It provides a structured overview of known reproducibility issues, particularly when dealing with LLMs, and presents the specific replication attempt in this work using the LIBRO framework with GPT-4o mini instead of the no longer available Codex model.

Relevance of Reproducibility The Association for Computing Machinery (ACM) defines reproducibility as follows [32]:

The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts.

This definition emphasizes not only technical traceability, but also the availability of models, data, and code.
Reproducibility is therefore a central principle of scientific research. It enables results to be verified, confidence in methods to be built, and progress to be made cumulatively, especially in dynamically growing, non-classically deterministic areas such as machine learning or LLMs. However, it is precisely in these fields that the principle faces particular challenges. As Kapoor and Narayanan [33] show, the problem of reproducibility affects not only computer science, but also disciplines such as chemistry, materials science, medicine, and climate research, where machine learning is increasingly being applied. The authors criticize that many ML-based studies are irreproducible due to erroneous or insufficiently documented training data, non-transparent model versions, or data leaks [33].
However, the increasing dependency on proprietary models such as OpenAI's Codex makes it difficult to independently reproduce experimental results. A key challenge is that these models are often not open source, so they can change without notice, or, as in the case of Codex, be discontinued altogether [34]. The associated implications for the reproducibility of scientific work have been highlighted by Angermeir et al. [35]. They show that many LLM-centric studies relying on OpenAI models have become difficult to verify. In their analysis of 86 ICSE 2024 and ASE 2024 papers, 18 studies provided research artefacts and used OpenAI models. They attempted to replicate these 18 studies and found that only five articles were sufficiently complete and executable. However, none of the five studies' results could be fully reproduced. Two studies appeared to be partially reproducible, while three did not seem reproducible at all [35].

Typical Challenges in Software Engineering In software engineering, outdated dependencies, incompletely documented environments, and a lack of standardization also pose major obstacles to reproducibility. A mapping study showed that the majority of

published work does not provide publicly accessible artifacts or clear instructions for replication [35]. Due to the lack of containerization or configuration management, many experimental setups are difficult to reconstruct. In addition, software changes rapidly due to updates and library changes, which makes replication under the same conditions even more difficult.

Additional Problems with LLMs LLMs also come with additional complications: models such as GPT-4 or Codex are subject to constant further development, are non-transparent, and are only accessible via closed APIs. This not only makes it difficult to access identical model states, but often prevents replication under the same conditions. Studies such as those by Chen et al. [36] show that the behavior of GPT-4 changed significantly between March and June 2023. For example, the success rate for mathematical tasks fell dramatically, even though the same model designation was used [36].
The strong prompt sensitivity and non-determinism of the models also pose problems. Szalontai et al. [37] found that results on the HumanEvalFix benchmark fluctuate greatly when decoding strategies, prompts, or model variants are slightly changed [37]. Dobslaw et al. [38] classify these difficulties in a taxonomy and show how multifaceted the test problems are in LLM-based systems, ranging from unpredictable behavior to input configurations that are difficult to test [38].

Reproducibility: LIBRO with GPT-4o mini Against this backdrop, this thesis positions itself as a concrete contribution to reproducibility research. The aim is to replicate the method described in the LIBRO paper. The same methodological setup is used, i.e. the same data, pipeline, evaluation metrics, and experimental design, but with a different model. For this work, GPT-4o mini (Snapshot: `gpt-4o-mini-2024-07-18`) is used instead of Codex. This model selection is necessary because Codex has not been available since March 23, 2023 [34]. This means that this reproduction differs in one single aspect, which may, however, be decisive.
Literature emphasizes the importance of conducting replication attempts even under slightly altered conditions. Works such as those by Angermeir et al. [35] explicitly call for reproducibility to be understood as part of the research process and for experimental statements to be critically examined for their transferability [35]. The present replication does exactly this by analyzing whether the results of the LIBRO paper can be repeated with a modern LLM or whether differences occur.

Significance and Outlook This thesis emphasizes that reproducibility is not just a theoretical ideal, but a practical challenge, especially in a dynamic research environment with changing models and dependencies. As Angermeir et al. [35] show, research urgently needs to develop robust standards and tools to systematically capture and promote reproducibility. These include open benchmarks, versioned APIs, standardization of prompt formats, and a stronger focus on artifact availability [35].
The ACM Badging Initiative [32] and the increasing number of replication studies,

such as HumanEvalFix, show that the community is increasingly facing up to this responsibility [37]. This thesis also sees itself as part of this discourse: it provides a documented, verifiable reproduction attempt that goes beyond mere application and addresses central questions of scientific traceability. The comparison of the results between Codex and GPT-4o mini also offers insights into model stability and the transferability of methods in the software engineering context.
Overall, it is clear that reproducibility, especially in the field of LLMs, is no longer just a matter of technical implementation, but increasingly also involves scientific and methodological dimensions. Such work helps to improve the transparency and robustness of research. This is a crucial step on the path to long-term valid, reliable findings.

5 Experimental Setup

This Section presents the experimental setup that forms the basis for answering the research questions in this thesis. The aim here is to present the experiments in a transparent and reproducible manner. To this end, the technical framework is described first, followed by the data sources used, the evaluation metrics, and the research protocol. Finally, the specific hardware and system environment on which the experiments were performed is explained.
The design of the experimental setup is directly based on the three research questions of this thesis, as already defined in Section 1:

RQ1 To what extent can the results presented in the LIBRO paper be reproduced?

RQ2 Which prompt components are particularly important for successful bug reproduction?

RQ3 Does adding additional information to the existing prompt components lead to improved results?

5.1 Data Sets

The significance of the experimental results depends largely on the quality and structure of the underlying data. In this work, data preparation includes both the extraction of bug reports and the generation of supplementary semantic context information in the form of Javadoc comments for the individual projects. Both data sources form the basis for the systematic construction of prompts and thus for the generation of test cases.

The Defects4J dataset is used as a benchmark, which is an established dataset for the empirical evaluation of software defects and automated testing methods. This dataset comprises a selection of 16 projects, including *Chart*, *Cli*, *Closure*, *Codec*, *Collections*, *Compress*, *Csv*, *Gson*, *JacksonCore*, *JacksonDatabind*, *JacksonXml*, *Jsoup*, *JxPath*, *Lang*, *Math*, *Mockito* and *Time*.
In addition, the GHRB dataset is used, which contains more recent and real-world bugs from open-source projects such as *Hakky54_sslcontext-kickstart*, *assertj_assertj-core*, *checkstyle_checkstyle*, *jhy_jsoup*, *google_gson* and *FasterXML_jackson-databind*. This combination enables evaluation based on both classic benchmark data and newer, practical bugs from GitHub repositories. An example of each of these bug reports from the data sets is shown in Listings 13 and 14.
Additionally, the number of individual bugs per project for both data sets is listed below:

Project	Bugs
Chart	8
Cli	37
Closure	174
Codec	18
Collections	4
Compress	47
Csv	16
Gson	18
JacksonCore	26
JacksonDatabind	112
JacksonXml	6
Jsoup	92
JxPath	22
Lang	64
Math	106
Mockito	38
Time	26
Total	**814**

Table 1: Overview of bugs per project of the Defects4J dataset.

Project	Bugs
AssertJ	5
checkstyle	13
Jackson	2
Gson	7
sslcontext	4
Jsoup	2
Total	**33**

Table 2: Overview of bugs per project of the GHRB dataset.

Each error instance consists of a buggy and a fixed commit as well as a natural language bug report. These bug reports are automatically parsed and structured so that they can be transferred as core components into prompt generation. Care is taken to ensure that the text fields (title, description, or stack trace, if available) are formatted uniformly to minimize tokenization effects in the LLM.

5.2 Experimental Design

The original LIBRO approach serves as a baseline against which all further experiments are compared. In addition, two further configurations are investigated: a reproduction variant that replicates the original prompt structure with a modern model, and a systematic ablation series in which the effect of individual prompt components is considered in isolation. The following table provides an overview of this technical setup:

Approach	Model	Prompt
LIBRO	Codex	Prompt used in the paper
LIBRO Reproducibility	GPT-4o mini	Reproducibility setup (identical prompt structure as in the original paper)
Systematic Prompting	GPT-4o mini	Ablation-based variations

Table 3: Overview of the technical setup for this work.

This overview forms the technical basis for analyzing the three research questions: to what extent the LIBRO results can be replicated (**RQ1**), which prompt components contribute significantly to successful bug reproduction (**RQ2**), and whether additional contextual information improves the quality of the results (**RQ3**).

Previous approaches to LLM-based test generation often use static, monolithic prompts that combine multiple instruction and context elements in a single text block [39, 40]. This work, on the other hand, takes a modular approach in which the prompt is broken down into clearly defined components. The goal is to empirically isolate and quantify the effect of these building blocks.
The basic building block of this work is the prompt used in the LIBRO approach, which basically consists of a bug report, a general instruction for creating a test, a code prefix, and a few example test methods (Few-Shot Learning). LIBRO integrates these components permanently, while this work systematically varies these components and additional documentation and examines them in different combinations. This allows us to analyze which information is crucial for successful reproduction, which components complement each other, and whether certain components are superfluous and redundant. Each prompt consists of five potential building blocks, which are shown in Table 4:

1. **Bug Report** – contains the natural language error description from the Defects4J and GHRB data sets. This error certificate ensures that the LLM has a textual basis that describes the error, its symptoms, and, if applicable, the expected fix.

2. **Command** – a standardized instruction that explicitly prompts the model to generate a JUnit test that reproduces the described error. This part serves to sharpen the focus of the LLM.

3. **Code Prefix** – a basic syntactic framework of the test class, consisting of an empty test method. This component gives the LLM structural orientation to avoid parser errors and better embed the generated code.

4. **Few-Shot Examples** – two exemplary bug report–test pairs from other projects that serve as learning examples for the model of what a successful bug reproduction looks like. This introduces the model to the desired output format.

5. **Documentation** – the Javadoc comments extracted from the projects, which provide additional semantic information about classes, methods, and API behavior. This component was introduced specifically as part of this work to reduce the LLM's knowledge gaps regarding project-specific implementation details.

Each of these components is specifically included or excluded in the ablation study in order to determine their individual influence on test generation.

Experiment	**Bug Report + Command**	**Code Prefix**	**Few-Shot Examples** ($n = 2$)	**Documentation**
LIBRO	✓	✓	✓	✗
LIBRO Reproducibility	✓	✓	✓	✗
Basic	✓	✗	✗	✗
Experiment 1	✓	✓	✗	✗
Experiment 2	✓	✗	✗	✓
Experiment 3	✓	✓	✗	✓
Experiment 4	✓	✗	✓	✓
All Ingredients	✓	✓	✓	✓

Table 4: Experimental setup with prompt component combination.

5.3 Experimental Configuration

Due to the partial non-determinism and randomness of the generation, which is typically modulated by weighted random sampling with a fixed temperature (e.g. 0.7 like in the paper) when querying the LLM, we cannot rely on a single evaluation run. Previous studies, such as the LIBRO investigation, used up to 50 generation attempts ($n = 50$) per bug report to achieve maximum reproduction performance. However, considering the higher costs associated with using OpenAI's model GPT-4o mini, we decided to significantly reduce the number of generation attempts. Consequently, we repeated the experiments only 5 times per bug and approach. The overall approach, analogous to LIBRO's procedure, comprised the following steps: (i) First, a prompt was created from the bug report. (ii) Then, 5 test candidates were generated per bug by querying the LLM. (iii) The generated tests were post-processed to resolve any remaining dependencies and make them executable. A strict timeout of 1 minute was enforced for the Defects4J build process, while GHRB builds were limited to 2 minutes because of its higher complexity. (iv) Finally, the results were selected and ranked based on their ability to fail in the faulty program.

Context Extraction Settings To ensure traceability and reproducibility, specific limits were defined for the documentation extraction process. The maximum length of extracted Javadoc sections was restricted by setting `MAX_DOC_CHARS` to 15000, and the number of permissible documentation blocks per class was limited to 15 via `MAX_BLOCKS_PER_CLASS`.

These constraints ensure that the context bundles remain information-rich yet prompt-efficient.
Furthermore, the inclusion of external libraries was activated (`ALSO_EXTERNAL=True`) to allow extraction from dependent `-sources.jar` archives located in the local Maven repository (`M2_REPO`). Detailed logging was enabled (`VERBOSE=True`) to facilitate debugging and replication.

Computational Setup The experiments were run on a *Dell R740xd* compute server with two *Intel Xeon 6254* processors (36 cores), 3.1GHz, 756GB of system memory, and a SuSE Leap 15 OS.

5.4 Evaluation Metrics

To evaluate the experiments in this work, a set of metrics is used to measure the success rate of bug reproduction and evaluate the ranking of the generated tests. These are the same metrics used in the paper by Kang et al. [1] to ensure better comparability of the results.
These metrics are:

1. **BRT (Bug Reproducing Test)**
   - A test is considered to be a BRT if it fails on the defective version (e.g. `PRE_FIX_REVISION` for Defects4J or Pre-merge for GHRB) and succeeds on the corrected version (e.g. `POST_FIX_REVISION` or Post-merge). A necessary condition for a BRT is that it compiles and fails in the faulty program (FIB test).
   - A bug is considered reproduced if the evaluation generates at least one BRT for that bug. The number of bugs that are reproduced is counted.

2. ***acc@n* (Accuracy at n)**
   This metric counts the number of bugs whose BRTs are found among the top–n rankings in the ranking list.

3. **Precision**
   Precision describes the proportion of bugs that were successfully reproduced among all bugs selected by LIBRO as potentially reproducible. It is calculated by dividing the number of bugs correctly reproduced by LIBRO by the total number of selected bugs. Precision thus serves as a measure of the reliability of LIBRO's selection strategy and indicates how often a recommendation by the system actually leads to successful bug reproduction.

4. ***wef@n*$_{agg}$**
   wef, *wef@n*, and *wef@*n_{agg} are interconnected metrics that quantify the manual testing effort caused by incorrectly prioritized tests. *wef* describes the wasted effort by counting how many non-reproducing tests are ranked above the first test that

actually reproduces the bug. *wef@n* extends this concept by looking at the wasted effort within the top n tests and indicating how many non-reproducing tests are in this limited upper ranking range. Finally, $wef@n_{agg}$ aggregates these values across all bugs considered and summarizes the total unnecessary inspection effort that arises when developers check the top-n candidates for each bug. $wef@n_{agg}$ thus forms an overall metric that shows how much manual work is caused on average by insufficient test prioritization.

6 Approach

This Section presents the approach taken in this work. It involves an expanded and systematic variant of the existing LIBRO framework, which makes it possible to specifically examine the effectiveness of individual prompt components in LLM-based bug reproduction. The core idea is to retain the original LIBRO workflow, but to supplement it with new mechanisms for systematic prompt engineering and automatic context enrichment. While LIBRO uses a fixed prompt template and OpenAI's Codex model, which is no longer available [34], the approach presented here relies on OpenAI's GPT-4o mini model and performs a controlled variation of the prompt components.
The goal of this work is to improve the reproducibility and explainability of existing methods. To this end, an ablation study is conducted in which individual prompt components (e.g. sample tests, code context, or documentation) are specifically added, removed, and combined to quantify their influence on the quality of the generated tests. In addition, specially developed Python scripts are used to extract relevant Javadoc information from the project sources and integrate it into the prompts to provide the model with additional semantic clues.
Figure 1 provides an overview of the process steps of the LIBRO workflow, which forms the basis of this work. This work starts at step A, prompt engineering, as explained above.

Overview

The extended approach of this work builds on the original LIBRO workflow and supplements it with mechanisms for systematic prompt engineering and automatic context enrichment through project documentation. The goal is to specifically investigate the role of individual prompt components in the process of LLM-based bug reproduction.

The complete workflow consists of five consecutive steps:

1. **Data Preparation**
   Extraction of bug reports from the Defects4J benchmark and collection of relevant project data. In addition, specially developed Python scripts are used to analyze and collect relevant Javadoc comments of the affected classes and store them in a structured JSON format.

2. **Prompt Engineering**
   The prompts are assembled from various components (bug report, command, code prefix, Few-Shot examples, optional documentation). These components are systematically added/left out as part of the ablation study.

3. **Test Generation**
   Transfer of the prompts to the LLM, which creates a JUnit test designed to reproduce the described bug.

4. **Test Execution**
   Execution of the generated tests in the LIBRO framework against the buggy and fixed versions of the tests.

5. **Evaluation and Analysis**
   Evaluation of the results based on the metrics defined in the LIBRO paper (see Section 5). The most effective combinations are identified by comparing the different prompt configurations.

These steps form the methodological basis of the extended LIBRO approach. The individual steps are described in more detail in the following sections of this Section.

6.1 Data Preparation

The significance of the experimental results depends largely on the quality and structure of the underlying data. In this work, data preparation includes both the extraction of bug reports and the generation of supplementary semantic context information in the form of Javadoc comments for the individual projects. Both data sources form the basis for the systematic construction of prompts and thus for the generation of test cases.
Each error instance consists of a buggy and a fixed commit as well as a natural language bug report. These bug reports are automatically parsed and structured so that they can be transferred as core components into prompt generation. Care is taken to ensure that the text fields (title, description, or stack trace, if available) are formatted uniformly to minimize tokenization effects in the LLM.

Extension through Semantic Context Since bug reports often contain only superficial and incomplete information [41], the LIBRO workflow in this work is extended by an additional semantic knowledge source: automatically generated Javadoc bundles. These serve as supplementary context in the prompt and are intended to enable the LLM to better establish the relationship between the error description and relevant API components.
To generate these Javadoc bundles, a separate extraction process was developed, consisting of two components:

1. `make_javadoc.py`[1]
   A script for generating a compact Javadoc bundle for a single project checkout.

2. `generate_all_docs.py`[2]
   A batch script that automates extraction across all selected projects and aggregates the results in a central JSON file.

The `make_javadoc.py` script retrieves project-internal metadata (e.g. source directory, classpath) via the Defects4J interface and then analyzes all Java files in the checkout. In

[1] https://github.com/martin-kunz/Master-Thesis/blob/main/scripts/make_javadoc.py
[2] https://github.com/martin-kunz/Master-Thesis/blob/main/scripts/generate_all_docs.py

the process, all import statements are collected, wildcards are resolved, and prioritized using a ranking heuristic to identify the classes most relevant to the respective bug report. This heuristic combines two weighting factors:

- Classes whose package corresponds to the project root package are given higher priority,
- and classes whose name appears lexically in the bug report are given additional weight.

Based on this weighted list, the corresponding Javadoc comments are extracted from the source code. If no project documentation is available, the script automatically searches linked `-sources.jar` archives in the local Maven repository to include external API documentation. The extracted text is then converted to a Markdown-like format, with each class represented as a section with a title (`## class name`) and the associated Javadoc blocks as quoted text blocks (`> ...`).
A typical extract from the generated JSON document is shown below:

```
{
  "Mockito": "# Library Docs Bundle for Mockito\n##
      org.mockito.AdditionalAnswers\n> Additional answers provides factory
      methods for less common answers.\n> <p>Currently offer answers that
      can return the parameter of an invocation at a certain position.\n>
      <p>See factory methods for more information: {@link
      #returnsFirstArg}, {@link #returnsSecondArg},\n> {@link
      #returnsLastArg} and {@link #returnsArgAt}\n> @since 1.9.5\n>
      @SuppressWarnings(\"unchecked\")\n> public class AdditionalAnswers {
      \n> private static final ReturnsArgumentAt RETURNS_FIRST_ARGUMENT =
      new ReturnsArgumentAt(0);\n> [...] "
}
```

Listing 1: Sample extract from an automatically generated Javadoc bundle for project *Mockito*. For better readability, the complete JSON content has been shortened. The complete files for all projects are available in the corresponding repository (see here).

Storing the documentation in this JSON format allows for lossless storage, and the structured representation allows the documentation segments to be inserted into the prompt in a targeted manner without losing the semantic context.

Data Organization To ensure traceability, all script parameters used in the documentation extraction process are versioned and documented. These include, in particular, the character and block limits (`MAX_DOC_CHARS`, `MAX_BLOCKS_PER_CLASS`), which define the maximum length of the extracted Javadoc sections and the number of permissible documentation blocks per class.
In addition, several control parameters are set that influence the runtime behavior and scope of the extraction. The variable `ALSO_EXTERNAL=True` activates the inclusion of

external libraries, thus allowing the extraction of additional documentation content from dependent `-sources.jar` archives. Although this option increases the runtime, it contributes significantly to the completeness of the semantic contexts. The parameter `VERBOSE=True` enables detailed logging of the extraction process, facilitating debugging, traceability, and subsequent replications. The variable `M2_REPO` refers to the system's local Maven repository (`/.m2/repository`), which serves as the search path for external source packages. This allows Javadoc comments from third-party libraries to be automatically located and included in the extraction.
The resulting JSON file is stored centrally and contains a complete documentation artifact for each project examined, limited according to the parameters defined above. This structured storage ensures that all experiments are reproducible, traceable, and can be re-executed under identical conditions.
This combination of targeted project selection, clearly defined extraction parameters, and semantic context enrichment creates a robust and traceable data basis. It forms the basis for the analysis described in the following sections, in which the effect of individual prompt components is quantitatively evaluated.

6.2 Prompt Engineering

The quality of the test cases generated by an LLM depends to a large extent on the quality and structure of the prompt used. Especially in the context of bug reproduction, the prompt design determines whether the model is able to correctly establish the semantic connection between a bug report and the code sections. This Section therefore focuses on the systematic investigation of the individual prompt components and their respective effects within the LIBRO workflow.

Conceptual Foundation Previous approaches to LLM-based test generation often use static, monolithic prompts that combine multiple instruction and context elements in a single text block [39, 40]. This work, on the other hand, takes a modular approach in which the prompt is broken down into clearly defined components. The goal is to empirically isolate and quantify the effect of these building blocks.
The basic building block of this work is the prompt used in the LIBRO approach, which basically consists of a bug report, a general instruction for creating a test, a code prefix, and a few example test methods (Few-Shot Learning). LIBRO integrates these components permanently, while this work systematically varies these components and additional documentation and examines them in different combinations. This allows us to analyze which information is crucial for successful reproduction, which components complement each other, and whether certain components are superfluous and redundant.

Prompt Structure and Components Each prompt consists of five potential building blocks:

1. **Bug Report** – contains the natural language error description from the Defects4J and GHRB data sets. This error certificate ensures that the LLM has a textual

basis that describes the error, its symptoms, and, if applicable, the expected fix.

2. **Command** – a standardized instruction that explicitly prompts the model to generate a JUnit test that reproduces the described error. This part serves to sharpen the focus of the LLM.

3. **Code Prefix** – a basic syntactic framework of the test class, consisting of an empty test method. This component gives the LLM structural orientation to avoid parser errors and better embed the generated code.

4. **Few-Shot Examples** – two exemplary bug report–test pairs from other projects that serve as learning examples for the model of what a successful bug reproduction looks like. This introduces the model to the desired output format.

5. **Documentation** – the Javadoc comments extracted from the projects, which provide additional semantic information about classes, methods, and API behavior. This component was introduced specifically as part of this work to reduce the LLM's knowledge gaps regarding project-specific implementation details.

6.3 Test Generation

In the test generation phase, the previously created prompts are transferred to the large language model GPT-4o mini, which automatically generates JUnit test methods based on the provided inputs. Generation is terminated as soon as the first occurrence of the string ‘‘‘ is detected, which marks the end of a code block.
The resulting tests are then syntactically cleaned up to remove any Markdown markup, incomplete code fragments, or missing import statements before being transferred to the execution step of the LIBRO framework.

6.4 Test Execution

To execute and validate the test methods generated by the LLM, a post-processing pipeline from the LIBRO approach is used. This process involves injecting the generated code into the existing test suite, resolving missing dependencies, and performing differential execution on the *buggy* and *fixed* versions of the project. Since two different data sets are used, the technical execution environment differs slightly, while the injection logic remains the same.

Test Injection and Dependency Resolving The raw test methods generated by the LLMs are now not executed in isolation, but are embedded in existing test classes in order to utilize the necessary context, such as helper methods, imports, and configurations. For each generated test method t_m, we will identify the most suitable test class C_{best} in the existing test suite T. This is done by means of a lexical similarity analysis. The idea here is that test methods that use similar identifiers and method calls require the same context.

The source file of C_{best} is read in and the generated method t_m is added. The code is checked for missing dependencies (unresolved references). This is resolved in two steps: The system first checks whether the required classes are already imported in C_{best} and can therefore be used without further adjustments. For all remaining missing types, the system then searches the entire project structure using `grep` for import statements whose path ends with the corresponding class name. The most frequently found import path is then selected and inserted into the file.
In addition, assertion libraries are normalized to avoid compatibility issues, especially for projects such as *jhy_jsoup* or *checkstyle_checkstyle* in the GHRB dataset.

Execution Environment After injection, the tests are compiled and executed. The pipelines for GHRB and Defects4J differ here due to the underlying build systems.

Execution for Defects4J For projects from the Defects4J benchmark, the provided `defects4j` command line tool is used to ensure a consistent environment.

- **Compiling:**
  Performed using `defects4j compile`. The output is parsed to extract specific `javac` error messages and filter out irrelevant warnings. These warnings include, for example, general compiler warnings (`warning:`), compiler notes (`[javac] Note:`), environment notes, or build tool metadata.

- **Test Run:**
  Tests are started in isolation with `defects4j test -t <TestName>`.

- **Timeout:**
  A strict timeout per test run is enforced by the timeout command to catch infinite loops in generated tests.

Execution for GHRB Since GHRB is based on real GitHub repositories, Maven (`mvn`) is used to control the build process.

- **Compiling:**
  The `mvn clean compile` command cleans the project of artifacts from previous builds and compiles the source code.

- **Test Run:**
  Execution is performed using `mvn test -Dtest=<TestName>`. To increase stability, additional flags are set for specific projects: `-Denforcer.skip=true` disables the Maven Enforcer plugin to avoid termination errors due to strict environment rules, while `-pl :sslcontext-kickstart` restricts execution to the relevant submodule in multi-module projects. In other cases, flags such as `-Djacoco.skip=true` prevent conflicts with code coverage tools or increase the error tolerance of the test suite using `-DfailIfNoTests=false`.

- **Timeout:**
  Due to the sometimes more complex build structures in GHRB, a longer timeout is used here.

Validation To ensure that a generated test actually reproduces the reported bug (bug reproducing test), a differential test ("two-version experiment") is performed. The injected test is executed on both the buggy version (V_{buggy}) and the fixed version (V_{fixed}). A test is considered successfully validated if the following conditions are met:

- **Buggy Version:**
  The test compiles successfully but fails. This indicates that the test triggers the bug in the code.
- **Fixed Version:**
  The test compiles and runs successfully. This confirms that the test is correct and does not fail due to its own logical errors.

Between executions, the status of the repository is completely reset using `git reset --hard` and `git clean -df` to prevent side effects from temporary files. In the case of the fixed version, the test code is synchronized from the buggy version to ensure that exactly the same test is evaluated on both versions.
A concrete example of such a successful validation run is shown in Listing 15 for run 3 of the *Codec-7* bug. The JSON log shows that the test listed in the `failed_tests`-list fails in the buggy version (V_{buggy}). The error message (`fib_error_msg`) precisely documents the cause: a `ComparisonFailure` in which the implementation incorrectly appended a line break to the Base64 string (visible in the diff between `expected:<...[ ]>` and `was:<... [\n]>`). In contrast, the list of failed tests (`failed_tests`) in the corrected version (V_{fixed}) is empty, confirming the success of the validation and the correctness of the fix.

6.5 Evaluation

Identifying tests that fail in the faulty version (FIB tests) is a necessary but not sufficient step. Not every FIB (Fail in Buggy program) test is a BRT (Bug Reproducing Test), as tests can also fail for trivial reasons, such as syntax errors that only occur at runtime.
To ensure the quality of the results and show developers only the most relevant tests, a two-step process is used: 1) Selection and 2) Ranking.

Selection The first step determines whether the results for a bug should be presented at all. This selection is based on the confidence of the LLM, which is measured by the consistency of the test results.
To do this, all FIB tests for a bug are grouped according to their error output, i.e. the same error type and the same error message. The idea here is that if several independently generated tests show exactly the same malfunction, it is likely that they reproduce the bug validly.

In the implementation, the size of the largest output cluster is compared to a set threshold. These clusters summarize all FIB tests that produce exactly the same error output when executed. The results are considered for the ranking only if the maximum match, i.e. the largest group of identical errors, exceeds this threshold.

Ranking Once a bug has been selected as confident, the associated FIB tests must be sorted to present the most likely BRT to developers first. Before ranking takes place, syntactically identical tests are filtered out first. Ranking is only performed on syntactically unique tests.
The ranking itself is based on three heuristics (in descending order of importance):

- **Agreement with the Bug Report:**
  Tests are prioritized whose error outputs, such as specific exceptions or incorrect values, or whose test code match the symptoms described in the bug report.
- **LLM Consensus:**
  The size of the output cluster is used as a secondary criterion. Greater agreement (consensus) is rated higher.
- **Test Length:**
  The final heuristic is test length, with shorter tests being preferred as they are easier to understand for developers.

Summary In summary, this Section has outlined the complete methodological pipeline of this work. This process begins with data preparation (Step 1), which includes Defects4J and GHRB and has been expanded to include a novel extraction of Javadoc contexts. Building on this, prompt engineering (Step 2) was explained in the context of an ablation study, and test generation (Step 3) was explained using GPT-4o mini. Subsequently, the Section on test execution (Step 4) detailed the technical implementation of the injection logic, dependency resolution, and differential validation (V_{buggy} vs. V_{fixed}) for both datasets. Finally, the evaluation strategy (Step 5) was explained, which ensures that only the most relevant and confident test results are used for analysis through a two-stage selection and ranking process.

7 Evaluation

This Section evaluates the performance and significance of our approach. To this end, we first conduct a reproducibility study to examine the extent to which the results proposed in the original study can be reproduced and confirmed. Next, the experimental results of the ablation study, which are based on the two datasets Defects4J and GHRB, are presented in order to evaluate the effectiveness of our approach. Finally, potential threats to internal and external validity that could influence the interpretability and generalizability of the results are discussed.
The reproducibility study also showed that some generated tests contained syntactic artifacts and therefore could not be compiled or executed. To evaluate these cases correctly, a separate script[3] was used to extract the test methods. The results in Section 7.1 are therefore considered both before and after this cleanup. In all subsequent experiments of the ablation study (*Basic*, *Experiment 1–4*, and *All Ingredients*), only the cleaned test cases were evaluated.

7.1 Reproducibility Study

In this Section, we examine the extent to which the results presented in the LIBRO paper can be replicated using our approach. The aim of this analysis is to evaluate whether GPT-4o mini, as a replacement for the Codex model originally used, achieves comparable performance in generating bug-reproducing test cases in order to answer our **RQ1**.
To answer this question comprehensively, we conduct the reproducibility study on two independent datasets, the Defects4J dataset and the GHRB dataset.

7.1.1 Results for Defects4J

In this Subsection, we present the results of the Defects4J dataset.

Experimental Results The experiments on the Defects4J dataset examine the reproducibility of the results from the original LIBRO paper. Since the original paper was based on Codex, a model that is no longer available, the reproducibility study was conducted here with GPT-4o mini. A total of three configurations were evaluated here. First, the results from the LIBRO paper, the reproduction of the experiments with GPT-4o mini, and a cleaned-up variant in which the generated tests were extracted from the generated test cases. Table 7 shows the complete results across all 16 Defects4J projects.
The results of the reproducibility study show large differences between the models. While LIBRO was able to reproduce a total of 251 out of 750 bugs in the original with Codex, the reproduction with GPT-4o mini only achieved 156 out of 731 reproduced bugs. This means that the reproduction rate of the GPT-4o mini-based approach is significantly

[3] https://github.com/martin-kunz/Master-Thesis/blob/main/scripts/clean_gen_tests.py

lower than the performance of Codex. The results remain consistent across all projects. In none of the larger or more complex projects such as *JacksonDatabind*, *Math*, *Lang*, or *Jsoup* does GPT-4o mini succeed in achieving the results of Codex.
Another difference can be seen when comparing direct reproduction with the cleaned-up variant. By extracting the generated test methods and removing artifacts created during generation, the number of reproduced bugs could be increased in some projects. Examples of this are Chart ($1 \rightarrow 2$), Compress ($4 \rightarrow 6$), *Gson* ($3 \rightarrow 4$), *JxPath* ($2 \rightarrow 4$), *Lang* ($17 \rightarrow 19$), and *Math* ($38 \rightarrow 45$). Overall, the adjusted variant increases the number of reproduced bugs from 156 to 176. Part of the difference between the unfiltered and adjusted variants is due to extraction artifacts, such as classes and imports, which were removed during the adjustment. However, the majority of the performance difference compared to Codex is still model-related.
The project-related distribution shows a heterogeneous pattern. Projects such as *Jsoup* or *Lang* also achieve comparatively high reproduction numbers (40 and 17–19, respectively) in the GPT-4o mini reproduction. Other projects, in particular *Mockito*, *JacksonXml*, and *JacksonDatabind*, remain difficult to reproduce. For *Mockito* and *JacksonXml*, GPT-4o mini fails to achieve a single reproduction in either the unadjusted or adjusted variant, even though the original LIBRO paper was able to successfully reproduce some of these bugs. This finding suggests that GPT-4o mini has difficulty understanding complex object interactions, internal state changes, and multi-step execution paths, which are typical in *Mockito* and *JacksonXml*. This assumption is supported by the error analysis of the original paper [1], which identified the lack of complex, project-specific helper methods as the most common cause of failure. In addition, tests in projects such as *JacksonXml* often fail due to dependencies on external files or resources that are referenced by the LLM in the code but cannot be created in the file system.
It is noteworthy that some projects show the opposite pattern. In *Compress*, *JxPath*, and *Math*, GPT-4o mini actually reproduces more bugs than Codex in the original LIBRO paper. For example, the cleaned-up version achieves 6 reproductions instead of 4 in *Compress*, 4 instead of 3 in *JxPath*, and 45 reproductions in *Math*, which is more than 43 reproduced with Codex. One reason for this behavior could be that certain bugs in these projects are primarily characterized by simple, clearly defined input-output relationships that are less based on complex object states or extensive library configurations. For such deterministic scenarios, a smaller model such as GPT-4o mini can be competitive despite its limited model capacity. In addition, the improvement achieved by the adjusted variant shows that a significant number of failed test cases are due to extraction or formatting artifacts and not to a lack of model capability.
Nevertheless, the overall picture here remains clear. The ability of GPT-4o mini to generate reproducible tests for Defects4J cannot replicate the performance of Codex. The moderate performance increase of the cleaned variant shows that careful post-processing of LLM-generated tests are necessary to reveal the actual potential of the models. The observed differences between projects such as *Jsoup* or *Math* on the one hand and *Mockito* or *JacksonXml* on the other underscore that the reproducibility of real bugs is highly dependent on the structural properties of the respective project and that small models perform significantly worse, especially with complex interaction patterns.

Project	LIBRO		LIBRO Repro.		LIBRO Repro. Cleaned	
	rep	**total**	**rep**	**total**	**rep**	**total**
Chart	**5**	**7**	1	7	2	7
Cli	**14**	**29**	10	29	12	29
Closure	2	172	**3**	**172**	2	172
Codec	**10**	**18**	8	18	8	18
Collections	1	4	1	4	1	4
Compress	4	46	4	46	**6**	**46**
Csv	6	16	6	16	6	16
Gson	**7**	**11**	3	11	4	11
JacksonCore	**8**	**24**	3	24	3	24
JacksonDatabind	**30**	**107**	10	107	11	107
JacksonXml	**2**	**6**	0	0	0	0
Jsoup	**56**	**92**	40	92	40	92
JxPath	3	19	2	19	**4**	**19**
Lang	**46**	**63**	17	63	19	63
Math	43	104	38	104	**45**	**104**
Mockito	**1**	**13**	0	0	0	0
Time	13	19	10	19	**13**	**23**
Total	**251**	**750**	**156**	**731**	**176**	**735**

Table 5: Reproduced bugs per project in the Defects4J dataset for *LIBRO*, the reproduction, and the cleaned variant. The columns rep and total denote the number of successfully reproduced bugs out of the total number of bugs for a given project.

Ranking Metrics The ranking metrics *acc@n* and *wef@*n_{agg}, as explained in Section 5.4, provide detailed insight into the quality and consistency of the generated test cases. The results show clear differences between the original LIBRO values and the reproductions with GPT-4o mini. The results are shown in Table 6.
Particularly noticeable is the decline in *acc@n* values for $n = 1$. While Codex achieves 0.43 in the original study, the values of the GPT-4o mini reproduction are 0.38 and 0.39 (cleaned variant). This means that reproduced tests appear significantly less often in first position in the ranking with GPT-4o mini. The model is therefore less precise in clearly prioritizing the most relevant inputs. However, for larger values of n, the differences even out. From $n = 3$ onwards, both the unadjusted and adjusted reproducibility variants are almost on par with the original results. These results show that GPT-4o mini does generate reproducible tests, but places them less reliably at the top of the ranking.
The *wef@*n_{agg} values show an even more nuanced picture. While the values for $n = 1$ are sometimes even slightly above the Codex results for GPT-4o mini, they decrease

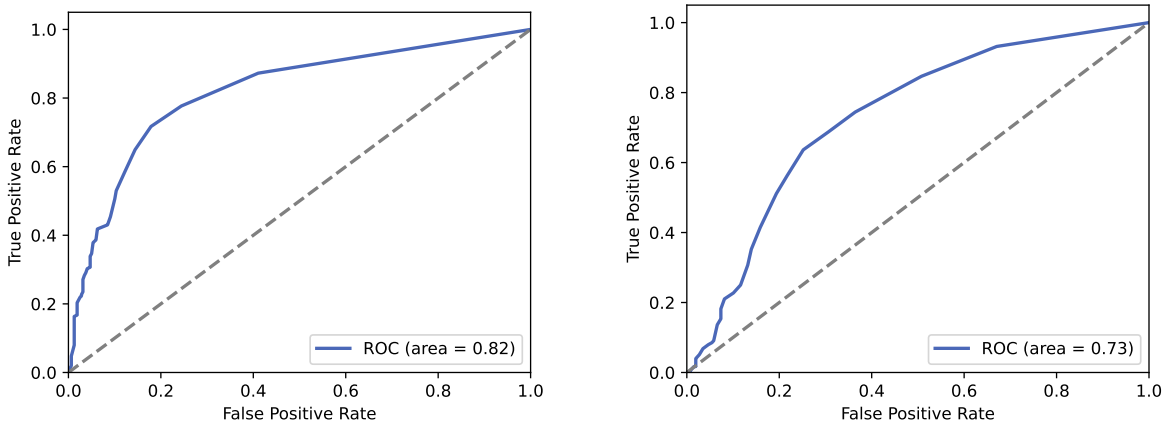
significantly as n increases. For $n = 5$, the GPT-4o mini reproduction only reaches 2.16 and 2.04 in the adjusted variant, while Codex achieves 2.28. This trend shows that GPT-4o mini generates individual strong error tests, but is less consistent across the entire test set. A plausible reason for this is that smaller models often generate exceptions that provide strong error signals but do not necessarily reflect the target bug. These effects can increase the $wef@n_{agg}$ values in the short term, but lead to lower consistency in larger test sets.
The cleaned variant systematically improves both $acc@n$ and $wef@n_{agg}$, albeit moderately. This suggests that some of the performance differences are due to structural artifacts of test generation, such as misplaced methods, syntactic fragments, class definitions, or import statements. The cleanup ensures that the test methods are executable and can be extracted correctly, which has led to additional reproductions in several projects. Nevertheless, the gap to the Codex results remains, making it clear that the central performance differences are predominantly model-related and do not result solely from formatting issues.
Overall, the ranking metrics confirm that GPT-4o mini can generate reproducible tests, but their positioning in the error ranking is less precise and their error patterns are less consistent than those of Codex. The adjusted variant mitigates these effects, but cannot compensate for them entirely. The results thus clearly show that the performance of the underlying model is crucial for the stability and prioritization of the test cases generated by the LLM.

	LIBRO	**LIBRO Repro.**	**LIBRO Repro. Cleaned**
		acc@n	
$n = 1$	**149 (0.43)**	80 (0.38)	100 (0.39)
$n = 3$	**184 (0.53)**	113 (0.54)	137 (0.54)
$n = 5$	**199 (0.57)**	119 (0.56)	145 (0.57)
		***wef@n*$_{agg}$**	
$n = 1$	201 (0.57)	**131 (0.62)**	156 (0.61)
$n = 3$	539 (1.54)	**325 (1.54)**	381 (1.49)
$n = 5$	797 (2.28)	**456 (2.16)**	522 (2.04)

Table 6: Performance comparison between LIBRO and reproduction experiments for the Defects4J dataset. Note: For $acc@n$, the values represent the total count of found bugs (precision); higher is better for both. For $wef@n_{agg}$, the values represent the total wasted effort (average per bug); lower is better for both.

ROC Curve The ROC curves (see Figure 2) complement the ranking metrics considered previously by evaluating the general discriminatory power between reproducing and non-reproducing tests. The AUC of the original LIBRO model (0.82) shows strong

discriminatory power. Codex thus generates error patterns that consistently and clearly differ from non-reproducing tests. The adjusted GPT-4o mini reproduction achieves a slightly lower but comparable value with an AUC of 0.73.
This suggests that GPT-4o mini is fundamentally capable of distinguishing reproducible tests based on their error signatures, but the consistency of these signals is lower than with Codex. This result thus also supports the observations from the values of $acc@n$ and $wef@n_{agg}$. It is worth noticing here that the AUC difference remains relatively small, even though the absolute reproduction rate is significantly lower. This can be explained by the fact that although GPT-4o mini generates fewer reproducing tests, these tests have strong error signatures that are easy to classify. Overall, the ROC analysis thus confirms a moderate weaker discriminatory power of GPT-4o mini.



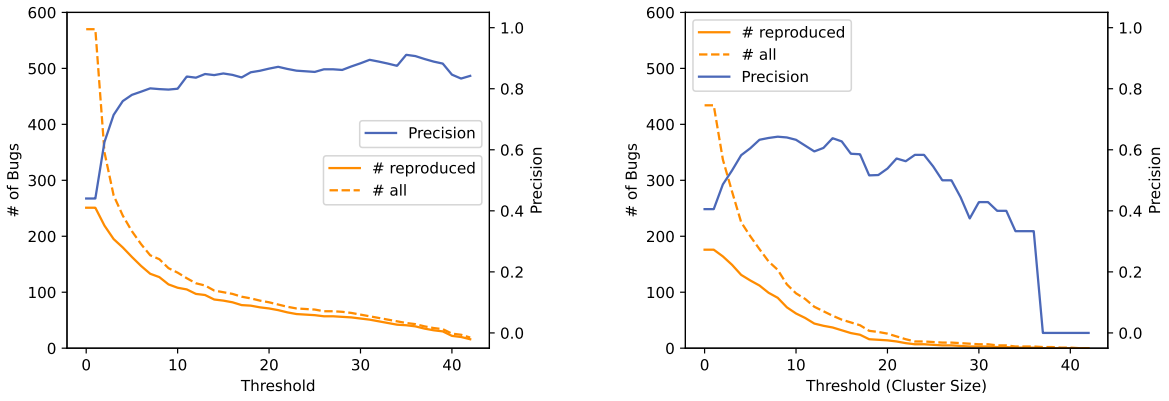
(a) Results for *LIBRO*.

(b) Results for *LIBRO Reproducibility Cleaned*.

Figure 2: ROC curve of bug selection.

Precision–Threshold Analysis The precision-threshold curves in Figure 3 show how the precision of the error tests changes when only test cases whose error severity exceeds a certain minimum threshold are considered. The precision of the original LIBRO results increases rapidly even at low thresholds and then stabilizes at a high precision level. This behavior indicates that Codex often generates clear error signatures and that reproducible tests tend to trigger strong errors that can be easily distinguished from other tests.
The reproducibility variant using GPT-4o mini exhibits a similar trend at low thresholds, where precision also increases quickly. However, in contrast to the Codex-based results, precision shows noticeable fluctuations in the mid-threshold range and does not converge to perfect precision for higher thresholds. Instead, precision decreases again for larger threshold values and eventually drops to near 0 as reproduced bugs disappear from the remaining candidate set. This indicates that fewer reproducing tests generated by GPT-4o mini are associated with consistently strong error severities, causing them to be

filtered out earlier as the threshold increases.
Overall, the precision–threshold analysis shows that GPT-4o mini follows the same general relationship between error severity and reproduction probability as Codex at low thresholds, but differs substantially at higher thresholds. While Codex maintains high precision across a broad threshold range, GPT-4o mini requires a careful balance between threshold selection and test coverage, as overly strict thresholds eliminate most reproducing tests. These results further support the observation that the primary limitation of GPT-4o mini lies in the consistency and robustness of the induced error signatures rather than in its ability to generate reproducing tests per se.



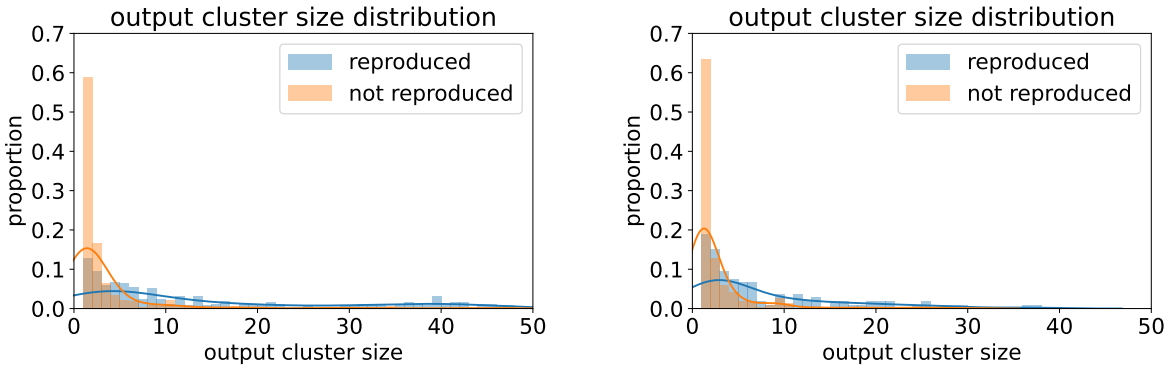
(a) Results for *LIBRO*.

(b) Results for *LIBRO Reproducibility Cleaned*.

Figure 3: Effect of thresholds to the number of bugs selected and precision.

Output Cluster Size Distribution The cluster size distributions in Figure 4 show how much the outputs of the generated tests vary and how this variation relates to reproducibility. A consistent pattern is visible in both variants, both in the original LIBRO paper and in the adjusted GPT-4o mini reproduction. On average, the reproduced bugs have larger and more widely distributed output clusters, while the non-reproduced bugs predominantly occur in very small clusters.
The noticeable peak at cluster size 1 for non-reproduced bugs shows that these tests often generate nearly identical error signatures, such as static exceptions or uninformative error outputs. Reproducible tests, on the other hand, are distributed over a significantly larger range (up to 50), suggesting that successful reproductions generate more complex and diverse error outputs originating from different execution paths. This pattern is nearly identical in both graphs. This shows that GPT-4o mini exhibits the same fundamental relationship between output heterogeneity and reproduction probability as the original Codex model.
The high similarity of the two distributions despite different model capacities shows that the size and diversity of error signatures are a robust indicator of the success of a reproducing test. Differences between the models are therefore less apparent in the shape

of these distributions and are more generated in the absolute number of reproducing tests.



(a) Results for *LIBRO*.

(b) Results for *LIBRO Reproducibility Cleaned*.

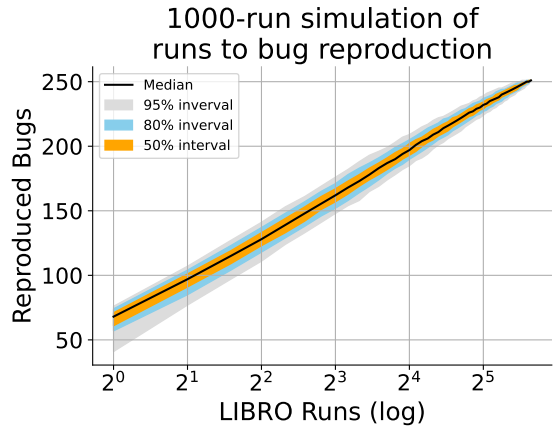
Figure 4: Distribution of the *max_output_clus_size* values for reproduced and not-reproduced bugs. For improved readability, the complete cluster results are provided in the accompanying repository (see here).

1000-Run Simulation Analysis The 1000-run simulations quantify how many reproductions can be expected when the number of sampling runs is increased. The four graphs (see Figure 5) show a very good log-linear relationship between the number of runs and the expected number of reproduced bugs. This confirms the central observation of the LIBRO paper. The probability of reproduction increases consistently and predictably with sampling intensity.

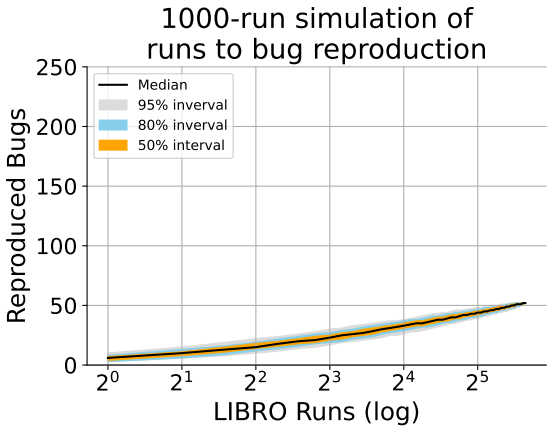
Comparing the original Codex simulations with the adjusted GPT-4o mini variant, it becomes clear that both models follow the same functional form of scaling (log-linear). However, contrary to the original results, the slope of the curves is significantly flatter for GPT-4o mini. While the confidence intervals remain comparable in their relative width, the steeper ascent in the Codex graphs indicates that the original model benefits significantly more from additional sampling runs. GPT-4o mini exhibits the same stochastic behavior type, but with a reduced efficiency gain per additional run.

The difference between the two experiments lies exclusively in the level of the curves. With an identical number of runs, the GPT-4o mini variant systematically achieves fewer reproductions. This applies to both bug reproduction and FIB (Fail in Buggy program) reproduction. The lower absolute level of reproductions reflects the limited model capacity of GPT-4o mini compared to Codex, but not a fundamentally different error distribution or sampling dynamics.

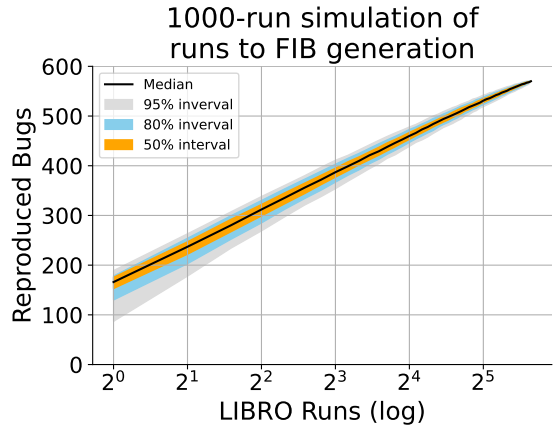
Overall, the simulations show that GPT-4o mini is less powerful but scales identically. More runs partially compensate for deficits, but do not change the underlying performance difference between the models. These results support the interpretation that the properties of the test generation process described in the LIBRO paper are robust and occur largely independently of the model used.



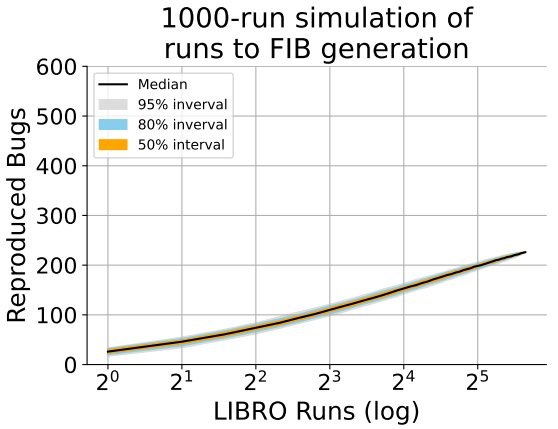
(a) Results for bug reproduction for *LIBRO*.



(b) Results for bug reproduction for *LIBRO Reproducibility Cleaned*.



(c) Results for FIB for *LIBRO*.



(d) Results for FIB for *LIBRO Reproducibility Cleaned*.

Figure 5: Generation attempts to performance for bug reproduction and for FIB.

Experimental Analysis The experiments performed on the Defects4J dataset show a consistent pattern across all evaluations examined. While the absolute number of reproduced bugs remains significantly below the original LIBRO paper results, the various metrics, such as *acc@n*, *wef@*n_{agg}, AUC, threshold precision, cluster sizes, and the 1000-run simulations largely reflect the same structural relationships observed in the LIBRO paper. In particular, the relationship between bug severity, output complexity, and reproduction probability shows a high degree of consistency between the two models, even though GPT-4o mini generates fewer reproducing test cases overall. Thus, the experiments reveal both clear quantitative differences and qualitative parallels, which together form the basis for the following answer to **RQ1**.

Partial Answer to RQ1 Based on the results presented above and the subsequent analyses, we have come to the following conclusion:

RQ1 The experiments on the Defects4J dataset show that the results reported in the LIBRO paper can only be partially reproduced with GPT-4o mini. Although the qualitative behavior of the system, such as the scaling of reproduction rates with the number of sampling runs and the characteristic patterns in the ranking and error distribution metrics, can be partially confirmed, the absolute reproduction performance lags significantly behind the original Codex model. The reproducibility study achieves only 156 reproduced bugs (or 176 after cleanup) compared to the 251 reproductions from the LIBRO paper. This drop in performance is consistent across almost all projects and illustrates that the original results were heavily dependent on the capabilities of the Codex model used at the time. Thus, the quantitative results of the LIBRO paper can only be reproduced to a limited extent.

7.1.2 Results for GHRB

In this Subsection, we present the results of the GHRB dataset.

Experimental Results The GHRB dataset comprises a total of 31 bugs from six Java projects. Table 7 shows the proportion of successfully reproduced bugs per project.
The results of the reproduction study clearly show that GPT-4o mini was only able to generate a valid bug-reproducing test case pair in a single case for the *Gson* project, while identifying all 31 bugs. This results in a success rate of 3.2% ($= \frac{1}{31}$) that is significantly lower than the 32.5% reported in the LIBRO paper for GHRB. A detailed analysis of the generated test methods shows that the vast majority of cases early failed due to syntactic errors.
After introducing the cleaning step that removes misplaced import statements and generated class declarations, we observe a modest improvement in the results. In the cleaned variant, GPT-4o mini successfully reproduced 2 out of 7 bugs in the *Gson* project, yielding an overall success rate of 6.5% ($= \frac{2}{31}$). This improvement indicates that the additional preprocessing step mitigates some early syntax failures and enables more test candidates to reach later stages of the LIBRO pipeline. Nevertheless, the overall performance remains far below the original Codex-based results.

Project	**LIBRO**		**LIBRO Repro.**		**LIBRO Repro. Cleaned**	
	rep	**total**	**rep**	**total**	**rep**	**total**
AssertJ	**3**	**5**	0	5	0	5
checkstyle	0	13	0	13	0	13
Jackson	0	2	0	2	0	2
Jsoup	**2**	**2**	0	2	0	2
Gson	**4**	**7**	1	7	2	7
sslcontext	**1**	**2**	0	2	0	2
Total	**10**	**31**	**1**	**31**	**2**	**31**

Table 7: Bug reproduction per project of the GHRB data set. The columns rep and total denote the number of successfully reproduced bugs out of the total number of bugs for a given project.

For further analysis, we distinguish between three categories of errors that are clearly evident in the GHRB dataset: **(A) Syntax Errors**, **(B) Semantic Errors**, and **(C) Runtime Errors**.

(A) Syntax Errors Of a total of 1587 generated test methods, 1251 led directly to an error message in the following form: `[error] JavaSyntaxError('')`. This means that

78.8% ($= \frac{1251}{1587}$) of all generated test methods could not be evaluated because they were rejected prior to compilation due to this syntactic error.
The error message occurred consistently in almost all projects and can be traced back to a pattern. The tests generated by GPT-4o mini do not correspond to the expected format of a single test method, but instead begin with import statements or a complete class definition, for example. The following example from the *AssertJ* project illustrates the problem:

```
1  import org.assertj.core.api.Assertions;
2  import org.junit.jupiter.api.Test;
3
4  class DefaultHasMethodTest {
5
6      interface HasDefault {
7          default void method() {}
8      }
9
10     static class Impl implements HasDefault {
11     }
12
13     @Test
14     void testHasMethod() {
15         Assertions.assertThat(Impl.class).hasMethod("method");
16     }
17 }
```

Listing 2: Example of a generated test case from *assertj_assertj-core_2324*, which cannot be injected into the expected LIBRO test method structure due to the class declaration and import statements it contains.

Although this code is syntactically correct, it violates the expected format, as LIBRO only injects individual methods (see Section 6.4). A complete class definition including imports will result in a syntax error at this point.
After cleaning up the test cases, this type of error is reduced to 1.1% ($= \frac{16}{1486}$). However, 101 ($1587 - 1486$) test cases are discarded because they have become completely empty or unusable after removing imports and class definitions (e.g. only comments remain, fragment code, etc.).

(B) Semantic Errors After many syntax errors are caught by removing interfering `import` or `class` blocks, semantic errors occur more frequently, i.e., errors that only become apparent during compilation.
The most common semantic errors can be briefly divided into three different error classes, which are presented below.

B.1 Name Conflicts with JUnit A particularly common pattern is the generation of classes named `Test`. If this class is injected into an existing test file, it conflicts with the JUnit annotation `@Test`.
Here is an example from *checkstyle*:

```java
public class Test {
    @Test
    public void testIndentationCheck() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 1000; i++) {
            sb.append(" ");
        }
        String longString = sb.toString();

        assertEquals(" ", longString.substring(0, 1));
    }
}
```

Listing 3: Example of a test case generated from *checkstyle_10825* that causes a name conflict with the JUnit annotation `@Test` due to a class declaration named `Test`.

This code is inserted as an inner class into a test class such as `XdocsPagesTest`. This causes `XdocsPagesTest.Test` to override the namespace of the annotation `@Test`. The compiler then attempts to interpret the annotation as an instance of the newly created class, which results in error messages of the form: `incompatible types: XdocsPagesTest.Test cannot be converted to java.lang.annotation.Annotation`.
These naming conflicts occurred in 5% ($= \frac{79}{1587}$) of all cases prior to the cleaned test cases. The additional preprocessing, which removes incorrect or inappropriately placed class definitions and imports, reduced this number, but the same problem still occurred in 2.7% ($= \frac{40}{1486}$) of all cases. This shows that even after cleanup, GPT-4o mini often generates complete or nested classes, meaning that the risk of name conflicts remains structurally.

B.2 Missing or fictitious Types The following example illustrates another common error:

```
/**
* Test method
*/
Test(E a) {
}
```

Listing 4: Example of a generated test case from *checkstyle_11926*, which leads to a compilation error due to a non-existent type (`E`).

Since `E` is not defined anywhere, we get: `cannot find symbol: class E`.
This category of errors occurred particularly frequently overall. Before cleaning, 8.4% ($= \frac{133}{1587}$) of the test cases generate this type of error. Although the additional cleaning significantly reduces the number of syntax errors, it also means that more test cases reach the compilation phase. As a result, the number of semantic compilation errors increases sharply. After cleaning, 54.4% ($= \frac{808}{1486}$) of all test cases generate this category of error. This development shows that GPT-4o mini often refers to non-existent types, methods, or classes, or produces incomplete signatures. After cleanup, category B.2 represents the most dominant source of errors within the entire GHRB evaluation.

B.3 References to non-existent Classes In many cases, GPT-4o mini generated test cases that referenced classes or methods that did not exist in the respective code base, as shown by the following error message: `AssertionError: Could not find titular class ColorWithCode ....`
While 2.7% ($= \frac{43}{1587}$) of the generated test cases are completely discarded before cleaning, this proportion increases to 14.2% ($= \frac{211}{1486}$) after cleaning. However, this effect is to be expected and can be explained directly by the objective of the cleaning step, since removing import statements and class definitions turns many generated outputs that were previously recognized as syntactically valid into empty or structurally incomplete test cases that do not contain any injectable test methods. In the original pipeline, such test cases often led to syntax errors within the Java parser, whereas in the cleaned pipeline they are filtered out early on. The increased rate of unusable test cases therefore does not represent a deterioration, but rather a desired refinement of the search space, as only test cases that actually represent a syntactically and structurally usable test method are passed on to the LIBRO pipeline. This shifts the error profile away from syntax errors to semantically relevant compilation errors.

(C) Runtime Errors A third category of errors concerns cases that do not involve syntactic or semantic compilation errors, but in which the test execution does not produce any usable output. This typically occurs in the event of timeouts or prematurely terminated Maven runs.

A typical example is:

```java
public void testClassFanOutComplexity() throws IOException {
    File myFile = new File("myfile.txt");

    try {
        InputStream stream = myFile.toURL().openStream();
    } catch (IOException | EmptyStackException e) {
        // Suppress exception
    }
}
```

Listing 5: Example of a generated test case from *checkstyle_10839*, which causes a runtime error after successful compilation.

This test compiles without errors and does not contain any assertions that could fail. Nevertheless, the pipeline reports: `compile_error = false`, `runtime_error = true` and `failed_tests = [ ]`.
This state occurs when the Maven process completes compilation but neither generates a `BUILD SUCCESS` nor a Surefire error (`<<< FAILURE!`). This typically happens when test execution is aborted, for example, due to the previously defined timeout of 2 minutes. This state is unusable for the ranking and selection logic, as no failure signature is generated.
In the original pipeline, runtime errors occurred in 2% ($= \frac{31}{1587}$) of cases. Cleaning slightly increases this percentage to 3.2% ($= \frac{48}{1486}$), as more test cases pass the syntactic analysis phase and thus enter the test execution phase.

Ranking Metrics The ranking metrics used in the LIBRO paper could not be calculated for any of the 31 bugs in the GHRB reproduction. Both *acc@n* and $wef@n_{agg}$ are equal to 0 for all n. Table 8 shows these results.
The reasons for these results are:

1. **Too few Executable Tests:**
   Most test cases fail before execution (categories (A) and (B)).

2. **No usable Failure Signatures:**
   Ranking is based on clustering of failure signatures. These are almost completely missing here.

3. ***acc@n* requires at least one successful Reproduction Test per Bug:**
   Since GPT-4o mini only generates a reproduction test for one or two bugs, it is not possible to generate statistically meaningful rankings.

This means that the selection and ranking results of the original LIBRO pipeline for GHRB cannot be evaluated in practice.

	LIBRO	LIBRO Repro.	LIBRO Repro. Cleaned
		acc@n	
$n = 1$	**6 (0.29)**	0 (0.00)	0 (0.00)
$n = 3$	**7 (0.33)**	0 (0.00)	0 (0.00)
$n = 5$	**8 (0.38)**	0 (0.00)	0 (0.00)
		wef@n$_{agg}$	
$n = 1$	**15 (0.71)**	0 (0.00)	0 (0.00)
$n = 3$	**42 (2.0)**	0 (0.00)	0 (0.00)
$n = 5$	**60 (2.86)**	0 (0.00)	0 (0.00)

Table 8: Performance comparison between LIBRO and reproduction experiments for the GHRB dataset. Note: For *acc@n*, the values represent the total count of found bugs (precision); higher is better for both. For *wef@*n_{agg}, the values represent the total wasted effort (average per bug); lower is better for both.

Answer to RQ1 Based on the results presented above and the subsequent analyses, we have come to the following conclusion:

RQ1 The reproduction of the results of the LIBRO paper is **only partially successful**. Although the experiments on Defects4J show that the basic mechanisms of the LIBRO pipeline, such as the relationship between defect severity, output complexity, and reproduction probability, can also be reproduced with GPT-4o mini, the quantitative performance lags significantly behind the original Codex-based results. On Defects4J, GPT-4o mini achieves only 156 reproduced bugs (176 after cleanup) compared to 251 in the original. The discrepancy is even more pronounced on the GHRB dataset, since Codex was able to reproduce 10 bugs, GPT-4o mini only manages 1 or 2 reproductions.
The analysis shows that this drop in performance is not due to the methodology itself, but primarily to the lower code generation quality of the model used. A large proportion of the generated test cases fail because of syntactic or semantic errors, which means that central pipeline components, such as the ranking metrics, are not effective. Qualitatively, the patterns of the LIBRO paper can thus be confirmed, but quantitatively, this is by no means the case. The results of the original paper cannot be reproduced to a comparable extent with GPT-4o mini.

7.2 Ablation Study

The following Section evaluates the results of the ablation study (see Table 4).
All experiments presented in this Section were cleaned as described above.

7.2.1 Results for Defects4J

The ablation study on the Defects4J dataset examines which prompt components are particularly relevant for successful bug reproduction. While the original LIBRO paper and our own reproducibility study each used 50 sampling runs per bug, the ablation experiments are based on 5 runs per bug and configuration. The results are shown in Table 9.
A consistent pattern emerges across all projects. The reduced variants *Basic* and *Experiment 1* to *Experiment 4* achieve only a fraction of the reproduction performance of LIBRO. Individual variants achieve moderate improvements, for example *Experiment 1* in *Math* (15 reproductions) or *Experiment 3* in *Cli* (4 reproductions). Overall, however, the results remain well below the performance of the complete prompt configurations.
The *All Ingredients* variant performs best. With 110 reproduced bugs, it achieves the highest success rate among all ablated variants and is significantly higher than the individual sub-configurations. This configuration combines all prompt components. The results show that additional contextual information, especially documentation and few-shot examples, significantly improve the performance of GPT-4o mini. Furthermore, it can be seen that all variants that are executed without a code prefix (*Basic*, *Experiment 2*, and *Experiment 4*) consistently achieve the worst results. This underscores the central importance of the code prefix for generating syntactically and semantically correct test cases, as it provides the model with clear structural orientation and thus contributes significantly to successful bug reproduction.
The project-related analysis also highlights significant differences between the Defects4J projects. Projects with clearly deterministic error conditions, such as *Jsoup*, *Math*, *Lang*, and *Csv*, benefit greatly from rich prompt configurations. It is worth noting that the *All Ingredients* variant reproduces even more bugs in individual projects such as *Closure* and *JxPath* than the *LIBRO Reproducibility* variant and, in some cases, the original Codex results. This suggests that additional contextual information can address certain types of errors better than the original prompt structure, even though the underlying model is weaker overall.
At the same time, the results also show clear limitations of GPT-4o mini's generative capabilities. In complex and nested projects such as *JacksonDatabind* or *Closure*, the absolute reproduction numbers remain low despite individual improvements, reflecting the high semantic complexity and object interdependence of these libraries. All ablation variants remain completely unsuccessful in *Mockito* and *JacksonXml*, two projects that were already among the most difficult cases in the original LIBRO paper. Here, the results suggest that the error mechanisms of these projects exhibit properties such as complex interactions, deep object graphs, or highly configuration-dependent error modes that cannot be reliably reconstructed even with advanced prompting strategies.

An interesting outlier is also *Experiment 1*, which, in contrast to all other ablated variants, achieves isolated reproductions in *Collections* and *Compress*, while the remaining ablated variants were unable to reproduce any bugs here. Overall, these project-related differences confirm that both the internal structure of a project and the type of error significantly determine the influence of individual prompt components on reproducibility.
Overall, the results show that while GPT-4o mini is fundamentally capable of generating bug reproductions even with reduced prompts, the combination of all prompt components achieves by far the highest reproduction rates. The *All Ingredients* configuration partially compensates for the limited model capacity and achieves the most robust results under the reduced conditions. The analysis of the ablation experiments thus makes it clear that complete and context-rich prompt structures are crucial for the reproduction performance of GPT-4o mini.

Project	Basic		Exp. 1		Exp. 2		Exp. 3		Exp. 4		All Ingredients	
	rep	total	rep	total	rep	total	rep	total	rep	total	rep	total
Chart	0	0	0	0	1	7	0	0	0	0	**2**	**7**
Cli	1	28	2	29	2	29	4	29	2	29	**5**	**29**
Closure	0	0	0	0	0	0	0	0	0	0	**4**	**172**
Codec	1	18	2	18	1	17	2	16	0	0	**4**	**18**
Collections	0	0	**1**	**4**	0	0	0	0	0	0	0	0
Compress	0	0	**1**	**46**	0	0	0	0	0	0	0	0
Csv	1	16	2	16	0	0	0	0	1	16	**4**	**16**
Gson	0	0	1	11	0	0	0	0	1	11	**4**	**11**
JacksonCore	1	23	**3**	**24**	2	21	1	24	2	24	2	24
JacksonDatabind	2	107	2	107	0	0	1	99	1	107	**7**	**107**
JacksonXml	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Jsoup	5	92	15	92	3	86	5	91	4	92	**32**	**92**
JxPath	0	0	1	19	0	0	0	0	0	0	**4**	**19**
Lang	3	63	**10**	**63**	3	58	3	54	4	63	9	63
Math	4	104	15	104	2	100	4	99	3	104	**27**	**104**
Mockito	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Time	3	23	3	23	3	23	2	23	3	23	**6**	**23**
Total	**21**	**474**	**57**	**545**	**17**	**341**	**22**	**435**	**21**	**469**	**110**	**685**

Table 9: Bug reproduction per project in Defects4J for the individual experiments. The columns rep and total denote the number of successfully reproduced bugs out of the total number of bugs for a given project.

Ranking Metrics The ranking metrics *acc@n* and *wef@n_{agg}* in Table 10 and 6 reveal a clear trade-off between the effectiveness (finding many bugs) and efficiency (minimizing developer effort) of the individual prompt configurations. As expected, the complete LIBRO paper results are significantly higher than the reduced variants in all cases, with the *LIBRO Reproducibility* configuration with GPT-4o mini also achieving high values, thus confirming the qualitative ranking behavior of the original paper. Among the ablated variants, *All Ingredients* consistently achieves the best results and is thus the only configuration that approximates the behavior of the complete prompt. This underscores that a rich prompt improves not only the absolute reproduction rate but also the ability to meaningfully prioritize reproducing test cases.

It is notable that some reduced variants achieve surprisingly high ranking scores despite significantly lower overall reproduction numbers. In particular, *Experiment 2* and *Experiment 3* achieve the highest precision (0.75) and the lowest wasted effort (*wef@n_{agg}* value of 2), even though they deliver only a few reproductions overall (6 and 9 respectively). This effect is due to the fact that a few strong error signatures within these variants appear particularly early in the ranking and are thus weighted disproportionately. At the same time, very low *wef@n_{agg}* values for the same variants show that the total number of such significant error signatures is low and that the models have difficulty generating consistent error patterns across multiple runs.

The *Basic* variant provides another interesting pattern. Despite low absolute reproduction numbers (21 overall), it sometimes achieves surprisingly stable ranking values, such as $wef@1_{agg} = 0.52$. This suggests that even minimally informed prompts occasionally generate test cases that trigger strong and clearly recognizable errors, even if this happens only rarely.

Overall, the ranking metrics clearly show that *All Ingredients* is the only ablated configuration that reliably and consistently approximates the complete LIBRO prompts. Reduced variants can achieve high scores in some cases, but these effects typically arise from a few strong outliers and do not reflect the overall quality of the prompt configuration.

	Basic	Exp. 1	Exp. 2	Exp. 3	Exp. 4	All Ingred.
			acc@n			
$n = 1$	12 (0.48)	24 (0.57)	9 (0.75)	6 (0.75)	16 (0.57)	**39 (0.48)**
$n = 3$	12 (0.48)	26 (0.62)	9 (0.75)	6 (0.75)	16 (0.57)	**49 (0.60)**
$n = 5$	12 (0.48)	26 (0.62)	9 (0.75)	6 (0.75)	16 (0.57)	**49 (0.60)**
			wef@n$_{agg}$			
$n = 1$	13 (0.52)	18 (0.43)	3 (0.25)	**2 (0.25)**	12 (0.43)	43 (0.52)
$n = 3$	30 (1.20)	39 (0.93)	8 (0.67)	**2 (0.25)**	23 (0.82)	94 (1.15)
$n = 5$	33 (1.32)	43 (1.02)	9 (0.75)	**2 (0.25)**	23 (0.82)	104 (1.27)

Table 10: Performance comparison between the ablation experiments for the Defects4J dataset. Note: For *acc@n*, the values represent the total count of found bugs (precision); higher is better for both. For *wef@*n_{agg}, the values represent the total wasted effort (average per bug); lower is better for both.

ROC Curve The ROC curve for the *All Ingredients* configuration (Figure 6) yields an AUC of 0.57, which is significantly lower than the values in the original LIBRO paper (AUC = 0.82). This means that the error signatures generated by the model are, at least in part, informative enough to prioritize reproducible tests over non-reproducible ones. Compared to the *LIBRO Reproducibility* variant, however, the curve is flatter, which is directly attributable to the lower consistency of the error signatures generated by GPT-4o mini. However, the AUC of the *All Ingredients* variant still lags significantly behind the Codex-based results of the original paper. This once again underscores that the quality of the ranking metrics is less limited by the prompt composition and more significantly by the capabilities of the underlying model.
For clarity, the other ROC curves for all remaining ablation variants (*Basic*, *Experiment 1–4*) can be found in the Appendix (Figure 10).

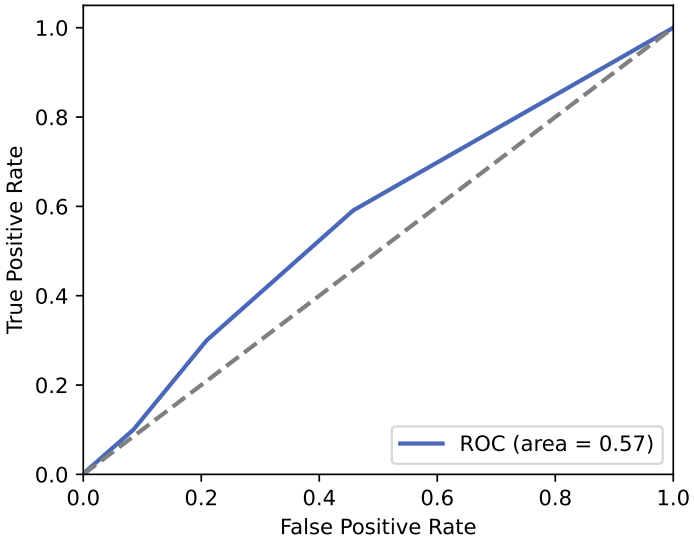


Figure 6: ROC curve of bug selection (Experiment *All Ingredients*).

Precision–Threshold Analysis The precision threshold curve for the *All Ingredients* configuration in Figure 7 deviates from the trends reported in the LIBRO paper. While precision remains constant in the low threshold range (cluster size 0–1) and rises slightly to approximately 0.5 in the middle range (cluster size 2–3), it exhibits inconsistent behavior at higher thresholds, with a drop at threshold 4. At the same time, the number of reproduced bugs decreases rapidly. At a threshold of 5, the number of remaining bugs approaches 0.

These results are primarily due to the methodological adjustment: Since the experiments were only performed with five sampling runs (instead of 50 as in the original configuration), the granularity of the cluster sizes is very coarse. The small sample size leads to high variance and data sparsity in the higher threshold ranges. A single false positive or true positive has a massive influence on the curve here, which explains the fluctuations at the end of the graph.

For clarity, the other Precision–Thresholds for all remaining ablation variants (*Basic*, *Experiment 1–4*) can be found in the Appendix (Figure 11).

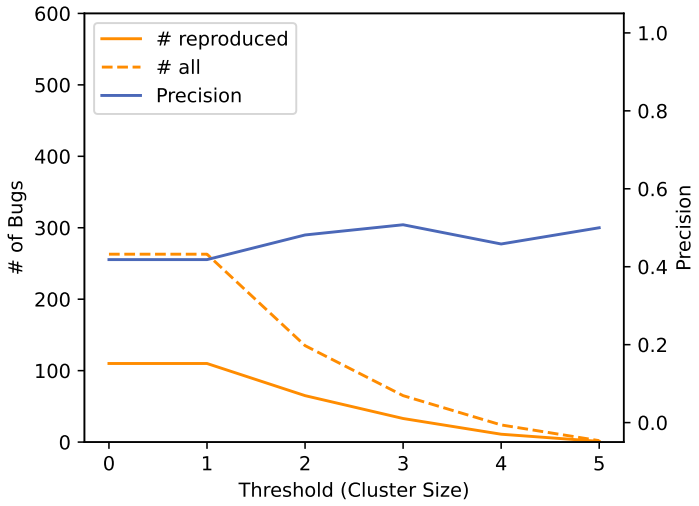


Figure 7: Effect of thresholds to the number of bugs selected and precision (Experiment *All Ingredients*).

Output Cluster Size Distribution The output cluster size distribution of the *All Ingredients* configuration (Figure 8) shows a significantly different picture than in the original LIBRO paper. While Codex typically generates few but distinct clusters with highly distinguishable error signatures, the outputs generated by GPT-4o mini are predominantly concentrated in very small cluster sizes in the range of 1 to 3. This applies to both reproduced and non-reproduced bugs. The density distributions of the two groups overlap almost completely, indicating that GPT-4o mini has difficulty consistently generating clear error behavior patterns. Reproducible tests thus differ little structurally from non-reproducible tests, which explains the low selectivity of the ranking metrics.
At the same time, it is noticeable that larger clusters, i.e., accumulations of similar error signatures, hardly occur. This contrasts sharply with the Codex results, where reproduced bugs often form dominant clusters that reliably guide the ranking. The flat, tightly concentrated distribution of the GPT-4o mini outputs thus shows that the model can generate syntactically executable tests, but hardly generates systematically distinguishable failure signatures. This finding is consistent with the results of the ROC curve (AUC = 0.62) and the $wef@n_{agg}$ values, both of which suggest a limited ability to prioritize meaningfully. Overall, the cluster analysis confirms that the weakness of the GPT-4o mini model lies less in the test generation itself and more in the lack of consistency and expressiveness of the generated error signals.
For clarity, the other Output Cluster Distribution for all remaining ablation variants (*Basic*, *Experiment 1–4*) can be found in the Appendix (Figure 12).

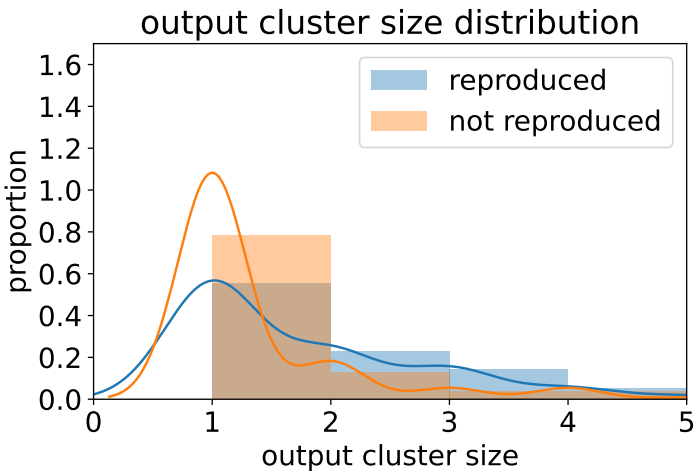
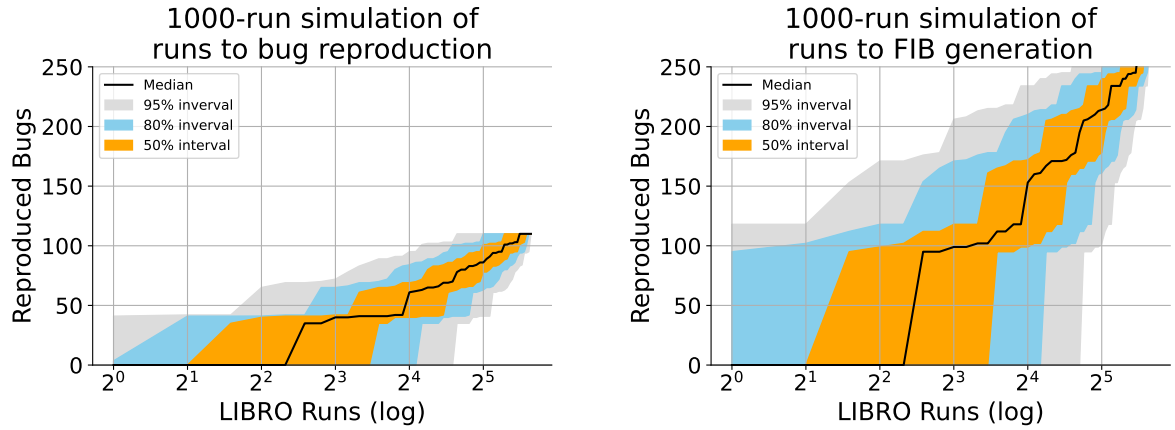


Figure 8: Distribution of the *max_output_clus_size* values for reproduced and not-reproduced bugs for the *All Ingredients* experiment.

1000-Run Simulation Analysis The 1000-run simulation for the *All Ingredients* configuration examines how many test cases need to be generated in order to achieve a certain reproduction performance (see Figure 9). Analogous to the procedure in the original LIBRO paper, a sample of test cases is repeatedly drawn for each bug, and this process is repeated 1000 times and evaluated. However, while LIBRO is based on 50 generated test cases per bug, only 5 test cases per bug are available in the present reproduction study with GPT-4o mini. As a result, the individual probabilities of success can only be estimated in rough increments (0, 0.2, 0.4, 0.6, 0.8, or 1.0). This limited granularity is directly transferred to the 1000-run simulation and leads to visible jumps and a less smoothed curve overall, which differs significantly from the original paper.
Despite this methodological limitation, the simulation shows a consistent pattern, as the median number of reproduced bugs increases steadily with the increasing number of LIBRO executions, and the confidence intervals expand steadily. The shape of the curve therefore confirms that additional test generations increase the probability of successful reproductions, even if the absolute performance upper limit of GPT-4o mini remains significantly below that of the originally used Codex model.
The other 1000 run simulations for all remaining ablation variants (*Basic*, *Experiment 1–4*) are shown in the Appendix (Figure 13).



(a) Generation attempts to performance for bug reproduction.

(b) Generation attempts to performance for FIB.

Figure 9: Generation attempts to performance for bug reproduction and for FIB (Experiment *All Ingredients*).

7.2.2 Results for GHRB

In this Section, we also evaluate the effectiveness of the various prompt components in the context of bug reproduction. While the previous Subsection presented a comprehensive ablation study under identical conditions based on the Defects4J dataset, the following analysis focuses on the GHRB dataset.
We also evaluate a total of eight prompt configurations: *LIBRO Reproducibility*, *Basic*, *Experiment 1*, *Experiment 2*, *Experiment 3*, *Experiment 4*, *All Ingredients*, and the results from the LIBRO paper, which serves as a reference. The results address both the reproduction rate per project and the metrics *acc@n* and *wef@*n_{agg}.

Experimental Results The results of the ablation study in Table 11 show a very clear pattern. With the exception of the *All Ingredients* configuration, none of the ablated variants reproduce a single bug. This applies to both the minimal variant *Basic* and the intermediate variants *Experiment 1* to *Experiment 4*. Only the combination of all components, including additional Javadoc documentation in *All Ingredients*, leads to reproduced bugs. Overall, 3 of the 31 bugs are successfully reproduced from the *Gson* project. This makes *All Ingredients* the most powerful configuration among the GPT-4o mini ablation based variants.

The results are particularly interesting in light of the fact that in the reproducibility study conducted in parallel, even the complete LIBRO prompt with GPT-4o mini and 50 sampling runs only achieved two reproduced bugs, also for *Gson*. In this context, the performance of *All Ingredients* with 3 reproduced bugs in only 5 sampling runs is particularly surprising. This observation suggests that additional prompt components such as Javadoc information for GPT-4o mini are indeed helpful and can at least partially

compensate for individual weaknesses of the model. At the same time, however, the absolute reproduction rate remains very low, which clearly underscores the limiting influence of the model.

The project-related distribution shows clear differences. Reproductions occur exclusively in *AssertJ*, *Jsoup*, and especially *Gson*. Projects such as *checkstyle*, *Jackson*, or *sslcontext*, on the other hand, show no reproducible cases, neither in the ablation variants nor in the LIBRO results. The reasons for this are project-structural and are also discussed in detail in the LIBRO paper. For *checkstyle*, the failure is due to the fact that the test cases are heavily dependent on external files that the LLM does not have access to, which prevents reproduction. In the case of *Jackson*, the small number of available bugs makes it difficult to make a reliable statement about why the experiments fail here as well [1].

Another special pattern can be seen in *Gson*. This project is the only one to show several successful reproductions, and in the reproducibility study, GPT-4o mini reproduces two bugs with the complete LIBRO prompt. These results can be explained by several factors. Unlike *checkstyle*, *Gson* is completely free of dependencies on external files. The tests operate exclusively on Java objects and JSON strings, so that the entire error logic runs within a closed, purely data-driven context. JSON processing is a deterministic transformation of inputs such as strings or objects into structured outputs and vice versa. This type of operation is largely free of race conditions, I/O dependencies, side effects, or state management, and can therefore be greatly simplified. For an LLM, this means that the error state to be reproduced can be reconstructed directly from the bug report using small inputs.
These project structural characteristics clearly distinguish *Gson* from libraries such as *Jackson* or *checkstyle*. While *Jackson* is a highly configurable and multifunctional serialization library whose malfunction often depends on complex object graphs or annotations, *Gson* is limited to simple, consistent processing patterns. *Checkstyle*, on the other hand, requires external configuration and XML files, which fundamentally prevents reproduction by LLMs.

Project	Basic		Exp. 1		Exp. 2		Exp. 3		Exp. 4		All Ingred.	
	rep	total	rep	total	rep	total	rep	total	rep	total	rep	total
AssertJ	0	5	0	5	0	5	0	5	0	5	0	5
checkstyle	0	13	0	13	0	13	0	13	0	13	0	13
Jackson	0	2	0	2	0	2	0	2	0	2	0	2
Jsoup	0	2	0	2	0	2	0	2	0	2	0	2
Gson	0	7	0	7	0	7	0	7	0	7	**3**	**7**
sslcontext	0	2	0	2	0	2	0	2	0	2	0	2
Total	**0**	**31**	**0**	**31**	**0**	**31**	**0**	**31**	**0**	**31**	**3**	**31**

Table 11: Bug reproduction per project in GHRB for the individual experiments. The columns rep and total denote the number of successfully reproduced bugs out of the total number of bugs for a given project.

Ranking Metrics In addition to the absolute number of reproduced bugs, the aggregated metrics introduced by LIBRO were also considered, including *acc@n* and *wef@*n_{agg}. The results in Figure 12 show that the ablation variants, including *All Ingredients*, have an acc@n and *wef@*n_{agg} value of 0.0 across all values of n (see Table 12). This contrasts with the results of the LIBRO paper, which achieved values between 0.29 and 0.38 (*acc@n*) and 0.71 and 2.86 (*wef@*n_{agg}) with Codex (see Table 8).
This deviation can be explained by the fact that although the ablated variants generate syntactically valid tests, these rarely or never fail in the manner described, or fail before or during compilation. As a result, the ranking metrics lack meaningful error signals to be included in the aggregation at all. Even *All Ingredients*, which successfully reproduces three bugs, remains ineffective in the global ranking metrics due to the low absolute number. These results once again confirm the central role of the model used. Only in combination with Codex is test generation stable and error-consistent enough to enable *acc@n* and *wef@*n_{agg} to provide meaningful measurements.

	Basic	**Exp. 1**	**Exp. 2**	**Exp. 3**	**Exp. 4**	**All Ingred.**
			acc@n			
$n = 1$	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
$n = 3$	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
$n = 5$	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
			***wef@*n_{agg}**			
$n = 1$	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
$n = 3$	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
$n = 5$	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)

Table 12: Performance comparison between the ablation experiments for the GHRB dataset. Note: For *acc@n*, the values represent the total count of found bugs (precision); higher is better for both. For *wef@*n_{agg}, the values represent the total wasted effort (average per bug); lower is better for both.

7.3 Experimental Analysis

Overall, the experiments on Defects4J and GHRB paint a consistent picture of the strengths and limitations of the LIBRO pipeline in combination with GPT-4o mini. First, many of the structural relationships reported in the original LIBRO paper can be qualitatively confirmed, as bugs with strong and diverse error signatures are more likely to be reproduced, reproduction rates increase monotonically with the number of sampling runs, and the ranking-based metrics *acc@n*, *wef@*n_{agg}, ROC curves, output cluster sizes, and 1000-run simulations show the same characteristic differences between reproduced and non-reproduced bugs. On the other hand, the quantitative performance gap with Codex-based results is significant and particularly pronounced in the GHRB

dataset, where GPT-4o mini reproduces only between 1 (*LIBRO Reproducibility*) and 3 bugs (*All Ingredients*) compared to 10 in the original approach of LIBRO.
The reproducibility study on Defects4J shows that GPT-4o mini is fundamentally capable of generating bug-reproducing tests, but does so less frequently and less reliably than Codex. The cleaned-up variant increases the number of reproductions from 156 to 176 and also improves *acc@n* and *wef@*n_{agg}. This suggests that a significant portion of the failures are due to syntactic and structural artifacts (e.g. misplaced import statements, class declarations, or fragmented methods) and not solely to a lack of model ability. At the same time, the gap to the original 251 reproductions remains significant. ROC analysis, precision–threshold curves, and output cluster distributions suggest that the error signatures generated by GPT-4o mini are qualitatively as informative as those generated by Codex. The difference lies primarily in the frequency and consistency with which such tests are generated.
The detailed error analysis for GHRB reinforces this picture. Here, the majority of the generated tests fail at the syntactic or semantic level, so that only a very small proportion ever reach the compilation, execution, and clustering phases. Error categories such as missing or fictitious types, incomplete method signatures, or references to non-existent classes dominate the profile after cleanup. As a result, hardly any usable error signatures are generated, and both *acc@n* and *wef@*n_{agg} are 0 for all n and therefore not measurable. The basic assumptions of the LIBRO pipeline, that the model reliably generates compilable tests with informative error templates, are largely violated for GHRB with GPT-4o mini. The stark contrast to Defects4J shows that dataset and project characteristics such as API complexity, configuration effort, or dependence on external files (e.g. in *checkstyle*) have a significant impact on the effectiveness of the approach.
The ablation study complements these findings by examining the influence of individual prompt components. Across both datasets, the reduced configurations (*Basic*, *Experiment 1–4*) achieve only a fraction of the performance of the full prompts. On Defects4J, the *All Ingredients* variant clearly stands out as the strongest ablation configuration, as it has 110 reproduced bugs and the best values for *acc@n*, *wef@*n_{agg}, ROC, and cluster distributions, it is the only variant that reliably approximates the behavior of the full LIBRO prompt. In several projects (including *Closure*, *JxPath*, *Gson*, *Jsoup*, *Math*), it even surpasses the *LIBRO Reproducibility* results and, in some cases, the original values, even though it is based on a weaker model and fewer runs. This suggests that additional contextual information, in particular documentation, code prefixes, and few-shot examples, can better address certain types of errors and partially compensate for model-related deficits. The consistently weak performance of all variants without code prefixes also underscores the relevance of these prefixes as explicit structural orientation for the generation of syntactically and semantically valid test methods.
This pattern is more evident on GHRB. None of the reduced variants reproduce a single bug here, and only *All Ingredients* achieves 3 reproductions, exclusively in the *Gson* project. On the one hand, this confirms that prompt design has a measurable effect, but on the other hand, it illustrates that its impact is strictly limited by the model capacity and structural properties of the target projects. Projects with closed,

purely data-driven error mechanisms such as *Gson* are significantly more accessible for LLM-based test generation than projects that rely heavily on external files, complex configurations, or deeply nested object graphs. Overall, the experiments show that the LIBRO pipeline is conceptually robust and that rich prompts systematically improve reproduction performance, but that the achievable upper limit of bug reproduction is ultimately determined by the interplay of model capability and project specifics, and not solely by the design of the prompt.

Answer to RQ2 and RQ3 Based on the results presented above and the subsequent analyses, we have come to the following conclusion:

RQ2 Successful bug reproduction is only possible through the interaction of several elements, besides the Bug Report and Command, with the **Code Prefix, Javadoc, and Few-Shot Examples** being particularly crucial. Variants without a Code Prefix consistently deliver the worst results, as the model lacks clear structural orientation. Similarly, the success rate increases noticeably as soon as additional contextual information is provided that precisely conveys the error-triggering conditions to the model. Few-Shot examples also reduce typical error formats and help the model learn from these examples and generate valid, executable test methods, resulting in more frequent reproduction tests overall. Overall, these three components are central to stable reproduction performance.

RQ3 The experiments clearly show that additional information in the prompt leads to significantly better results. With 110 reproduced bugs, the ***All Ingredients* configuration achieves the best performance of all ablation variants** and even surpasses *LIBRO Reproducibility* in a few projects and, in some cases, the original values of the LIBRO paper. Additional documentation and expanded contextual knowledge support GPT-4o mini especially in projects with deterministic error conditions and partially compensate for model weaknesses. The ranking metrics also improve consistently. This shows that rich, well-structured prompt information significantly increases the effectiveness of test generation and is therefore a key prerequisite for successful reproduction performance.

8 Discussion and Limitations

The results of this work clearly show that language models such as GPT-4o mini are fundamentally capable of generating bug reproduction tests from natural language descriptions. However, their performance depends heavily on the design of the prompt and the complexity of the target project. Systematic analysis of the prompt components shows that successful bug reproduction only occurs through the interaction of several elements. It appears that the code prefix provides necessary syntactic orientation, contextualizing information conveys the required semantics, and few-shot examples reduce incorrect formats and increase the probability of valid test methods. This observation is consistent with results from related studies, which also show that LLM-based software tools are highly sensitive to prompt design, contextual richness, and model capacity [42–44].
At the same time, it becomes clear that the performance of GPT-4o mini is insufficient to reproduce the results of the original LIBRO paper, which is based on the Codex model. Particularly complex projects, such as *Mockito*, *JacksonXml*, and large parts of *Closure*, could not be reproduced in any ablation variant. These projects are characterized by factors that have been shown to complicate LLM-based test generation, including reflection-based APIs, deeply nested object graphs, high context dependency, and atypical test infrastructures [43–45]. Such structures require precise semantic modeling and stable execution paths that smaller LLMs cannot reliably reconstruct. Similar challenges are also described in recent benchmarks for LLM code understanding, particularly in SWE-Bench-Java and MultiOOP, where models systematically fail on semantically challenging bugs [43, 44].
The study by Wang, S., et al. [43] also explicitly shows that GPT-4o mini exhibits a particularly sharp performance drop between 16.21% and 19.84% on the MultiOOP benchmark for Java-based tasks [43]. There, performance drops significantly in Java analysis and API interactions, supporting the conclusion that smaller models have difficulty correctly capturing complex object relationships and dynamic execution contexts. This result is directly consistent with the observations in this work, as the projects that rely on intensive object manipulation, reflection-based calls, or complex build environments are precisely those in which GPT-4o mini was unable to achieve a single reproduction.
The results also show that, despite poorer absolute reproduction rates, many qualitative patterns from the original LIBRO paper could be confirmed, such as the increasing sampling runs of reproduced bugs, reproducing tests exhibiting more complex error signatures, and ranking metrics reflecting more consistent error characteristics. This leads to the conclusion that the pipeline logic of LIBRO is robust, with the limitation primarily lying in the underlying model.

Threats to Validity

This Subsection summarizes the internal and external validity threats and limitations of this thesis.

Internal Validity This paragraph presents the internal and external threats to validity of this thesis.
LLMs such as GPT-4o mini exhibit non-deterministic behavior, meaning that identical inputs may yield different outputs depending on sampling processes, internal model stochastics, hardware execution paths, and undisclosed inference implementations. Even with restrictive decoding settings such as *temperature = 0* and *top-k = 1*, a residual degree of randomness remains. To ensure comparability with the original LIBRO paper, the same sampling parameters, including a temperature of 0.7, were adopted.
Another internal validity concern is the limited number of runs. Although multiple executions were conducted per configuration, the total number of repetitions is lower than the 50 runs reported in the original LIBRO paper, reducing statistical robustness and increasing sensitivity to random fluctuations.
Furthermore, only a small number of test cases were generated per bug ($n = 5$), which restricts the granularity of the subsequent 1000-run simulation. This results in coarse outcome distributions (0, 0.2, 0.4, ..., 1.0) and causes some curves to appear jumpy or less smooth compared to the original paper.
Finally, methodological design choices introduce additional internal threats. Prior work has shown that the ordering of prompt information can significantly influence LLM performance. For example, reversing input order in defect localization tasks has led to substantial drops in top-1 accuracy [46]. Since only a single prompt structure was evaluated, alternative prompt formulations might yield different results.

External Validity External validity concerns the generalizability of the results to other models, datasets, programming languages, and software projects.
The evaluation is intentionally restricted to Java projects from Defects4J and GHRB, which may not be representative of other ecosystems with different architectural patterns, languages, or testing infrastructures. Moreover, the original LIBRO paper relies on Codex, a model with different capacity, training data, and architectural properties. Consequently, observed performance differences cannot be attributed unambiguously to the proposed method alone, but may also be influenced by model-specific characteristics.
Model selection constitutes another limitation. GPT-4o mini is smaller than Codex and has been shown in recent studies to be weaker at code generation and complex software understanding [42, 43]. Particularly for the GHRB dataset, the low reproduction performance may therefore be partly explained by limited model capacity rather than methodological shortcomings.
Additionally, several target projects exhibit structural properties that are inherently challenging for LLM-based systems. Projects such as *Mockito* or *JacksonXml* rely heavily on reflection or highly configurable pipelines, which increase error space complexity and hinder reliable test reproduction [44]. In *Closure*, an atypical test infrastructure further complicates the extraction and integration of generated test methods. As a result, the findings cannot be fully generalized to projects with similar characteristics.
Another external validity threat arises from potential overlap between the model's training data and the evaluated projects. GPT-4o mini has a knowledge cutoff in October 2023 [47],

and both Defects4J and GHRB were publicly available before that date. While exposure to these codebases cannot be ruled out, both this work and related studies indicate that prior exposure does not automatically translate into improved reproduction performance, especially for complex software systems [5, 6].
Finally, only a single LLM was evaluated. Other models explicitly trained on code, such as StarCoder2 or Code Llama, may achieve better results [48]. Therefore, the conclusions are limited to the evaluated model and should not be generalized to LLMs in general.

9 Conclusion and Future Work

This work systematically investigated the extent to which the results of the LIBRO approach are reproducible, which prompt components are crucial for successful bug reproduction by LLMs, and whether adding additional components and context, Javadoc of the affected classes in this work, improve the results.
The reproduction studies on Defects4J and GHRB consistently show that while GPT-4o mini can replicate the qualitative behavior of LIBRO, the absolute success rates lag significantly behind the original Codex results. In particular, complex projects such as *Mockito*, *JacksonXml*, or large parts of *Closure* remained completely unreproducible with GPT-4o mini, while more simply structured libraries such as *Gson* allowed for more frequent successful reproductions.
The ablation study clearly shows that the performance of the model depends heavily on the design of the prompt, as only the combined interaction of bug report, request to generate a test method, a code prefix, Javadoc documentation, and Few-Shot examples lead to stable reproductions of bugs, while reduced variants achieve poorer performance. Nevertheless, the results clearly show that additional contextual information in the form of relevant Javadoc helps, but cannot compensate for the fundamental model limitations of smaller LLMs such as GPT-4o mini. Overall, the work illustrates that both model capacity and prompt design remain key success factors for LLM-based test generation and that the reproducibility of LIBRO depends significantly on the codex model originally used.

Future Work Based on the results of this work, several promising directions for future research and work emerge.
One obvious approach is to systematically vary the order and structure of the prompt components, as previous work shows that the order and sequencing of information can significantly influence the performance of LLMs [46]. Similarly, the use of more code-specialized and open source models such as StarCoder2, Code Llama, or DeepSeek-Coder-V2 seems reasonable to investigate whether models trained specifically for programming tasks exhibit greater robustness toward complex Java projects. These models would also address the problem of deprecation of models such as Codex described in the previous Sections, as these are open models [49–51]. In addition, the ablation experiments should be repeated with a larger number of sampling runs to determine more robust performance profiles. Further research could also expand syntactic and semantic preprocessing steps to reduce the high error rate of generated tests, as well as investigate dynamic context enrichment, for example through incremental prompting or retrieval-based support with project-specific examples. In the long term, a comparison with emerging multi-agent systems or tool-based LLM frameworks such as OpenHands [48] or SWE-Agents [44] would also be promising in order to evaluate whether different prompting strategies for agent systems can achieve a higher success rate than a single model.
Overall, the work shows that the field of LLM-based bug reproduction is still far from reaching its full potential and offers numerous open research questions, especially at the interface between model architecture, prompt design, and test infrastructure.

References

[1] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.

[2] René Just, Darioush Jalali, and Michael 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*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Jae Yong Lee, Sungmin Kang, Juyeon Yoon, and Shin Yoo. The GitHub Recent Bugs Dataset for Evaluating LLM-Based Debugging Applications. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 442–444, 2024.

[4] Laura Plein, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models. ICSE-Companion '24, pages 360–361, New York, NY, USA, 2024. Association for Computing Machinery.

[5] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc.

[6] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[7] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[8] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic Bug Reproduction for Effective Automated Program Repair at Google, 2025.

[9] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating Tests from Issues to Validate SWE Patches. *arXiv preprint arXiv:2502.05368*, 2025.

[10] Irtaza Sajid Qureshi and Zhen Ming (Jack) Jiang. Test Case Generation from Bug Reports via Large Language Models: A Cognitive Layered Evaluation Framework. *arXiv preprint arXiv:2510.05365*, 2025.

[11] Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F Bissyandé, Yang Liu, and Haoye Tian. A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications. *arXiv preprint arXiv:2506.23749*, 2025.

[12] Jiajun Sun, Fengjie Li, Xinzhu Qi, Hongyu Zhang, and Jiajun Jiang. Empirical Evaluation of Large Language Models in Automated Program Repair. *arXiv preprint arXiv:2506.13186*, 2025.

[13] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A Comprehensive Overview of Large Language Models. *ACM Trans. Intell. Syst. Technol.*, 16(5), August 2025.

[14] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications, 2025.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[16] Steven Davies and Marc Roper. What's in a bug report? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2014.

[17] Oracle. How to Write Doc Comments for the Javadoc Tool. https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html. Accessed: 11 October 2025.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[19] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations (ICLR)*, 2020.

[20] Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. Is Temperature the Creativity Parameter of Large Language Models? In *International Conference on Computational Creativity*, April 2024.

[21] Lujun Li, Lama Sleem, Niccolo' Gentile, Geoffrey Nichil, and Radu State. Exploring the Impact of Temperature on Large Language Models: Hot or Cold?, 2025.

[22] Moin Nadeem, Tianxing He, Kyunghyun Cho, and James Glass. A Systematic Characterization of Sampling Algorithms for Open ended Language Generation. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 334–346, Suzhou, China, December 2020. Association for Computational Linguistics.

[23] Phil McMinn. Search-Based Software Test Data Generation: A Survey. Software Testing, Verification & Reliability. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[24] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[25] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[27] Phil McMinn. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. USENIX Association.

[29] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated Whitebox Fuzz Testing. In *Ndss*, volume 8, pages 151–166, 2008.

[30] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, January 2025.

[31] Berk Atil, Sarp Aykent, Alexa Chittams, Lisheng Fu, Rebecca J. Passonneau, Evan Radcliffe, Guru Rajan Rajagopal, Adam Sloan, Tomasz Tudrej, Ferhan Ture, Zhe Wu, Lixinyu Xu, and Breck Baldwin. Non-Determinism of "Deterministic" LLM Settings, 2025.

[32] Association for Computing Machinery (ACM). Artifact Review and Badging - Current. https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2020. Accessed: 13 November 2025.

[33] Sayash Kapoor and Arvind Narayanan. Leakage and the Reproducibility Crisis in ML-based Science, 2022.

[34] OpenAI. Deprecations. https://platform.openai.com/docs/deprecations. Accessed: 2 November 2025.

[35] Florian Angermeir, Maximilian Amougou, Mark Kreitz, Andreas Bauer, Matthias Linhuber, Davide Fucci, Daniel Mendez, Tony Gorschek, et al. Reflections on the Reproducibility of Commercial LLM Performance in Empirical Software Engineering Studies. *arXiv preprint arXiv:2510.25506*, 2025.

[36] Lingjiao Chen, Matei Zaharia, and James Zou. How Is ChatGPT's Behavior Changing over Time? *Harvard Data Science Review*, 6(2), 2024.

[37] Balázs Szalontai, Balázs Márton, Balázs Pintér, and Tibor Gregorics. Investigating Reproducibility Challenges in LLM Bugfixing on the HumanEvalFix Benchmark. *Software*, 4(3), 2025.

[38] Felix Dobslaw, Robert Feldt, Juyeon Yoon, and Shin Yoo. Challenges in Testing Large Language Model Based Software: A Faceted Taxonomy. *arXiv preprint arXiv:2503.00481*, 2025.

[39] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.

[40] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. An Empirical Study of Unit Test Generation with Large Language Models. *arXiv preprint arXiv:2406.18181*, 2024.

[41] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. Association for Computing Machinery.

[42] Konstantinos Vergopoulos, Mark Niklas Mueller, and Martin Vechev. Automated Benchmark Generation for Repository-Level Coding Tasks. In *Forty-second International Conference on Machine Learning*, 2025.

[43] Shuai Wang, Liang Ding, Li Shen, Yong Luo, Han Hu, Lefei Zhang, and Fu Lin. A Multi-Language Object-Oriented Programming Benchmark for Large Language Models. *arXiv preprint arXiv:2509.26111*, 2025.

[44] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. *arXiv preprint arXiv:2408.14354*, 2024.

[45] Gregory Gay. Challenges in Using Search-Based Test Generation to Identify Real Faults in Mockito. In *International Symposium on Search Based Software Engineering*, pages 231–237. Springer, 2016.

[46] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Order Matters! An Empirical Study on Large Language Models' Input Order Bias in Software Fault Localization, 2025.

[47] OpenAI. OpenAI Model Docs. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 21 November 2025.

[48] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741*, 2024.

[49] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The Next Generation, 2024.

[50] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*, 2023.

[51] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou,

Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence, 2024.

Appendix

```
[
    {
        "role": "system",
        "content": "You are an assistant for augmenting the test suite in
            our Java software. You write a JUnit test case that reproduce
            the failure behavior of the bug report."
    },
    {
        "role": "user",
        "content": "Generate a self-contained reproducing test case from
            the following bug report.\n{%examples/1_query.txt%}"
    },
    {
        "role": "assistant",
        "content": "{%examples/1_response.txt%}"
    },
    {
        "role": "user",
        "content": "Generate a self-contained reproducing test case from
            the following bug report.\n{%examples/2_query.txt%}"
    },
    {
        "role": "assistant",
        "content": "{%examples/2_response.txt%}"
    },
    {
        "role": "user",
        "content": "Generate a self-contained reproducing test case from
            the following bug report.\n{{bug_report_content}}"
    }
]
```

Listing 6: Prompt template of the original LIBRO paper, which was also used for the *LIBRO Reproducibility* series of experiments.

```
[
    {
        "role": "system",
        "content": "You are an assistant for augmenting the test suite in
            our Java software. You write a JUnit test case that reproduce
            the failure behavior of the bug report."
    },
    {
        "role": "user",
        "content": "Generate a self-contained reproducing test case from
            the following bug report.\n{{BUG_REPORT_CONTENT}}"
    }
]
```

Listing 7: Prompt template for the test series *Basic*.

```
[
    {
        "role": "system",
        "content": "You are an assistant for augmenting the test suite in
            our Java software. You write a JUnit test case that reproduce
            the failure behavior of the bug report."
    },
    {
        "role": "user",
        "content": "Generate a self-contained reproducing test case from
            the following bug
            report.\n{{BUG_REPORT_CONTENT}}\n\n---IMPORTANT: Your entire
            response must be *only* the Java code for the test method. Start
            your response directly with the method signature ‘public void
            test‘."
    }
]
```

Listing 8: Prompt template for the test series *Experiment 1*.

```
[
  {
    "role": "system",
    "content": "You are an assistant for augmenting the test suite in our
       Java software. You write a JUnit test case that reproduce the
       failure behavior of the bug report."
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n\n{{BUG_REPORT_CONTENT}}\n\n---\n\nTo help
       you, here is some relevant API documentation for the project
       '{{PROJECT_NAME}}':\n---\n{{JAVADOC_DOCUMENTATION}}"
  }
]
```

Listing 9: Prompt template for the test series *Experiment 2*.

```
[
  {
    "role": "system",
    "content": "You are an assistant for augmenting the test suite in our
       Java software. You write a JUnit test case that reproduce the
       failure behavior of the bug report."
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n\n{{BUG_REPORT_CONTENT}}\n\n---\n\nTo help
       you, here is some relevant API documentation for the project
       '{{PROJECT_NAME}}':\n---\n{{JAVADOC_DOCUMENTATION}}\n\n---\nIMPORTANT:
       Your entire response must be *only* the Java code for the test
       method. Start your response directly with the method signature
       `public void test`."
  }
]
```

Listing 10: Prompt template for the test series *Experiment 3*.

```
[
  {
    "role": "system",
    "content": "You are an assistant for augmenting the test suite in our
       Java software. You write a JUnit test case that reproduce the
       failure behavior of the bug report."
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n{%examples/1_query.txt%}"
  },
  {
    "role": "assistant",
    "content": "{%examples/1_response.txt%}"
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n{%examples/2_query.txt%}"
  },
  {
    "role": "assistant",
    "content": "{%examples/2_response.txt%}"
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n\n{{BUG_REPORT_CONTENT}}\n\n---\n\nTo help
       you, here is some relevant API documentation for the project
       '{{PROJECT_NAME}}':\n---\n{{JAVADOC_DOCUMENTATION}}"
  }
]
```

Listing 11: Prompt template for the test series *Experiment 4*.

```
[
  {
    "role": "system",
    "content": "You are an assistant for augmenting the test suite in our
       Java software. You write a JUnit test case that reproduce the
       failure behavior of the bug report."
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n{%examples/1_query.txt%}"
  },
  {
    "role": "assistant",
    "content": "{%examples/1_response.txt%}"
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n{%examples/2_query.txt%}"
  },
  {
    "role": "assistant",
    "content": "{%examples/2_response.txt%}"
  },
  {
    "role": "user",
    "content": "Generate a self-contained reproducing test case from the
       following bug report.\n\n{{BUG_REPORT_CONTENT}}\n\n---\n\nTo help
       you, here is some relevant API documentation for the project
       '{{PROJECT_NAME}}':\n---\n{{JAVADOC_DOCUMENTATION}}\n\n---\n
       IMPORTANT: Your entire response must be *only* the Java code for
       the test method. Start your response directly with the method
       signature `public void test`."
  }
]
```

Listing 12: Prompt template for the test series *All Ingredients*.

```
{
    "issue_id": "785",
    "title": "anonymous object type inference inconsistency when used in
       union",
    "description": "Code:\r\n/** @param {{prop: string, prop2:
       (string|undefined)}} record */\r\nvar func = function(record) {\r\n
       window.console.log(record.prop);\r\n}\r\n\r\n/** @param {{prop:
       string, prop2: (string|undefined)}|string} record */\r\nvar func2 =
       function(record) {\r\n if (typeof record == 'string') {\r\n
       window.console.log(record);\r\n }else {\r\n
       window.console.log(record.prop);\r\n }\r\n}\r\n\r\nfunc({prop:
       'a'});\r\nfunc2({prop: 'a'});\r\n\r\n\r\n\r\n\r\n\r\nerrors
       with:\r\nERROR - actual parameter 1of func2 does not match formal
       parameter\r\nfound :{prop: string}\r\nrequired: (string|{prop:
       string, prop2: (string|undefined)})\r\nfunc2({prop:
       'a'});\r\n\r\n\r\n\r\nthe type of the record input to func and func2
       are identical but the parameters to func2 allow some other type."
}
```

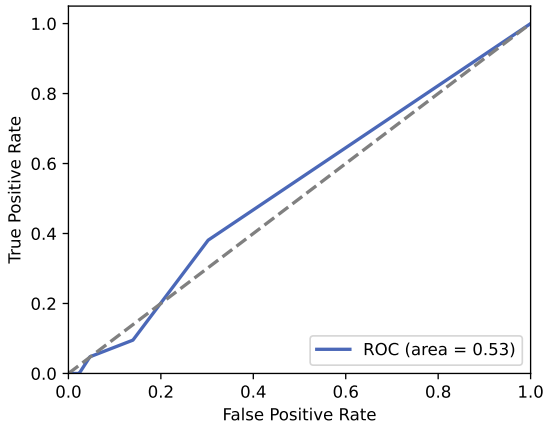

Listing 13: Sample bug report for *Closure-166* from the Defects4J dataset.

```
{
  "issue_id": 121,
  "issue_url": "https://github.com/Hakky54/sslcontext-kickstart/issues/121",
  "title": "Remove/Disable logging in unsafe HostnameVerifier and
     TrustManager",
  "description": "<p dir=\"auto\">the logging in the unsafe variants of
     HostnameVerifier and TrustManager spam logs. There should be a way to
     disable this or at least log it to DEBUG.</p>\n<p dir=\"auto\">I see
     no reason why i would like to know that a self signed certificate is
     trusted. that's exactly what i want to do else i wouldn't use this
     specific verifier.</p>",
  "description_text": "the logging in the unsafe variants of
     HostnameVerifier and TrustManager spam logs. There should be a way to
     disable this or at least log it to DEBUG.\nI see no reason why i
     would like to know that a self signed certificate is trusted. that's
     exactly what i want to do else i wouldn't use this specific verifier."
}
```

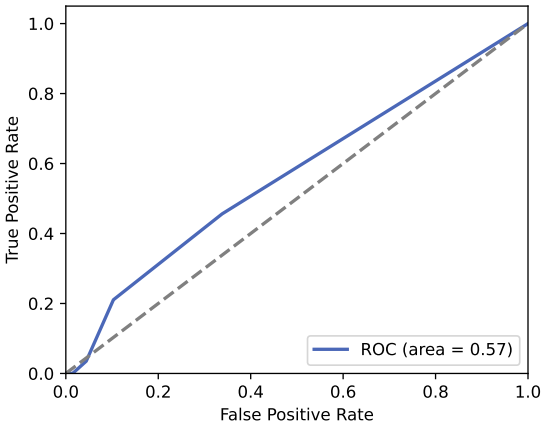

Listing 14: Sample bug report for *Hakky54_sslcontext-kickstart-122* from the GHRB dataset.

```
{
    "Codec_7": {
        "Codec_7_n3.txt": {
            "buggy": {
                "compile_error": false,
                "runtime_error": false,
                "failed_tests": [
                    "org.apache.commons.codec.binary.Base64Test::
                        testEncodeBase64StringDoesNotChunkAutoGen"
                ],
                "autogen_failed": true,
                "fib_error_msg": "---
                    org.apache.commons.codec.binary.Base64Test::
                    testEncodeBase64StringDoesNotChunkAutoGen\n
                    junit.framework.ComparisonFailure: expected:<Zm9vYmFy[]>
                    but was:<Zm9vYmFy[\n]>\n\tat
                    junit.framework.Assert.assertEquals(Assert.java:100)\n
                    \tat junit.framework.Assert.assertEquals(Assert.java:107)
                    \n",
                "compile_msg": null
            },
            "fixed": {
                "compile_error": false,
                "runtime_error": false,
                "failed_tests": [],
                "autogen_failed": false,
                "fib_error_msg": null,
                "compile_msg": null
            },
            "success": true
        }
    }
}
```

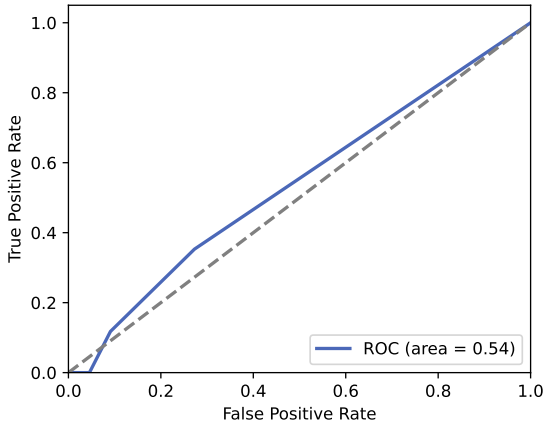

Listing 15: Example of a successfully validated run, here using Run 3 of the *Codec-7* bug.



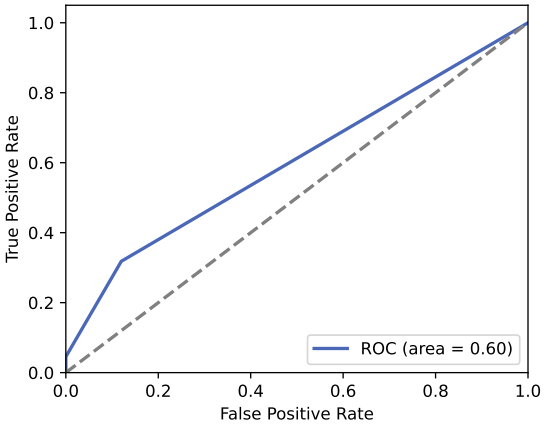
(a) Results for the test series *Basic*.



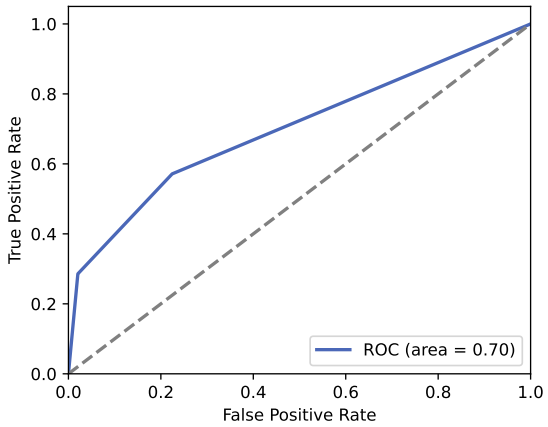
(b) Results for the test series *Experiment 1*.



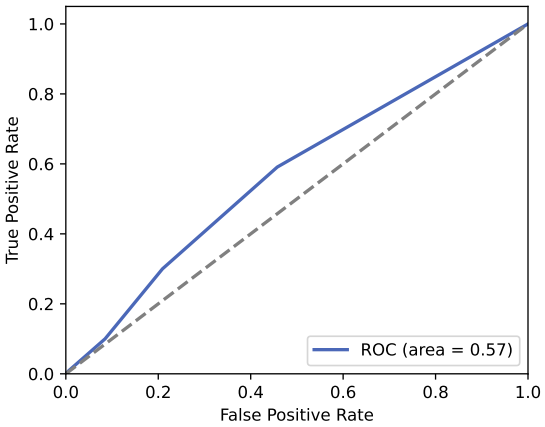
(c) Results for the test series *Experiment 2*.



(d) Results for the test series *Experiment 3*.

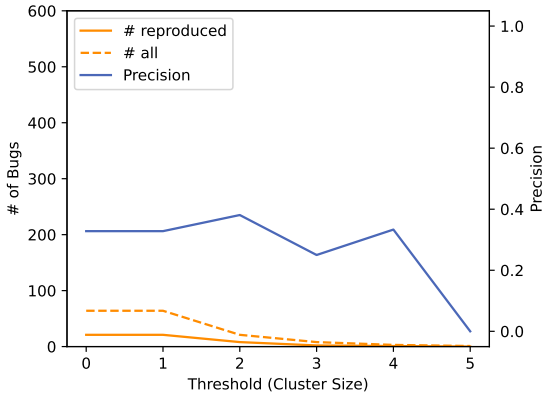


(e) Results for the test series *Experiment 4*.

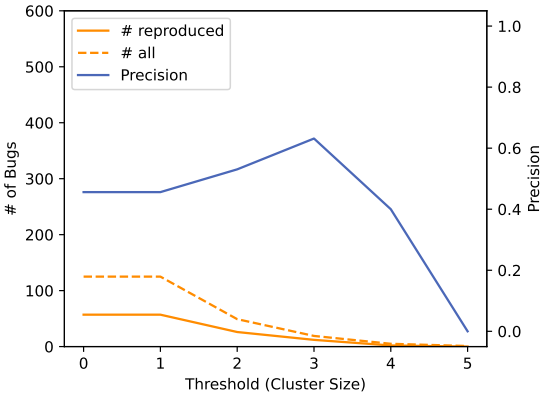


(f) Results for the test series *All Ingredients*.

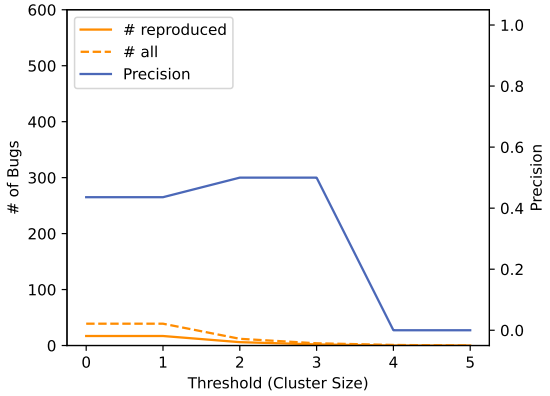
Figure 10: ROC curve of bug selection for the Defects4J dataset.



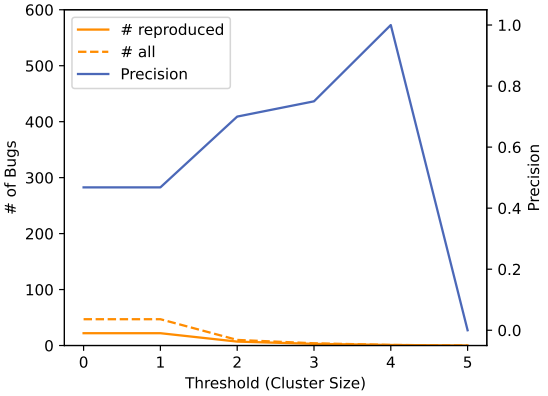
(a) Results for the test series *Basic*.



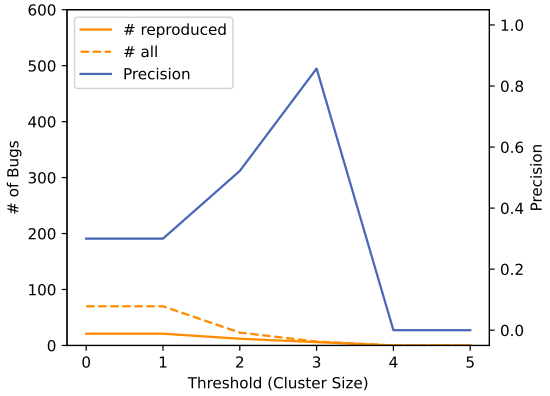
(b) Results for the test series *Experiment 1*.



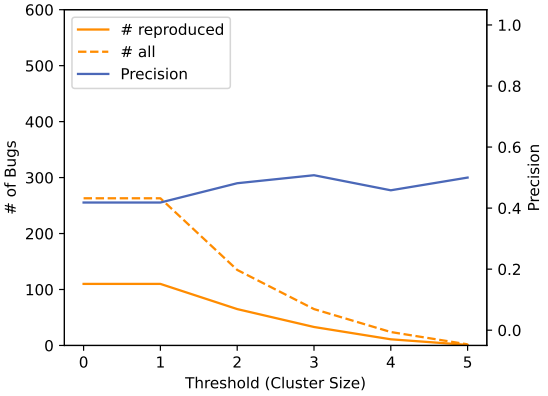
(c) Results for the test series *Experiment 2*.



(d) Results for the test series *Experiment 3*.

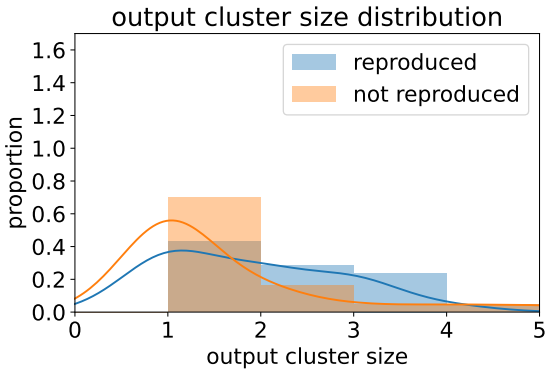


(e) Results for the test series *Experiment 4*.

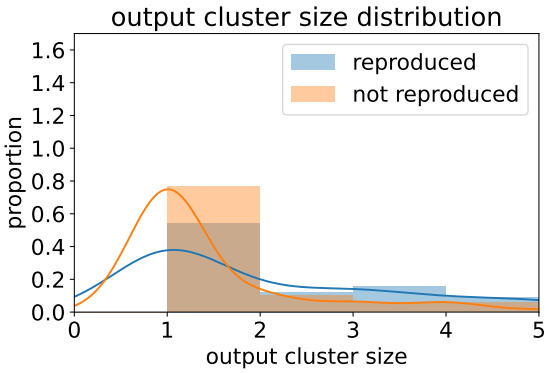


(f) Results for the test series *All Ingredients*.

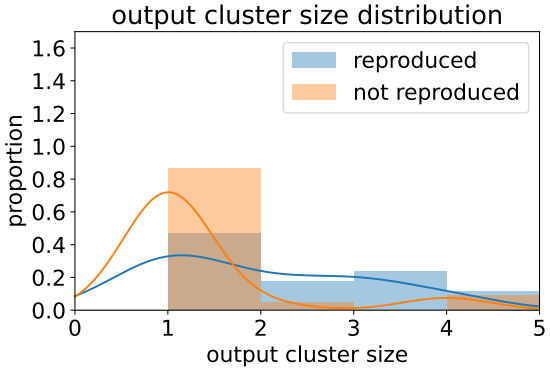
Figure 11: Effect of thresholds to the number of bugs selected and precision for the Defects4J dataset.



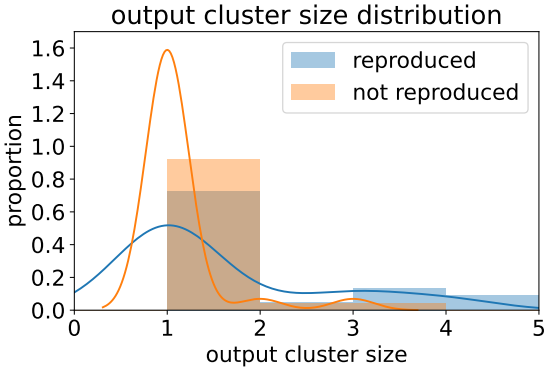
(a) Results for the test series *Basic*.



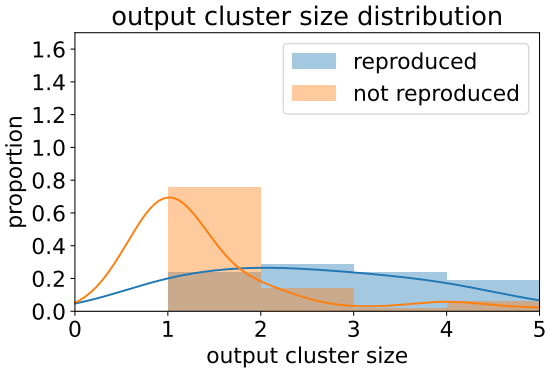
(b) Results for the test series *Experiment 1*.



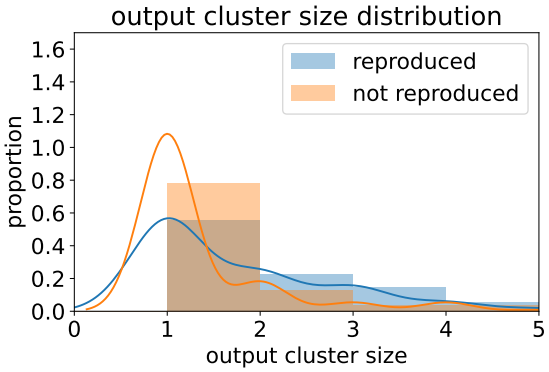
(c) Results for the test series *Experiment 2*.



(d) Results for the test series *Experiment 3*.

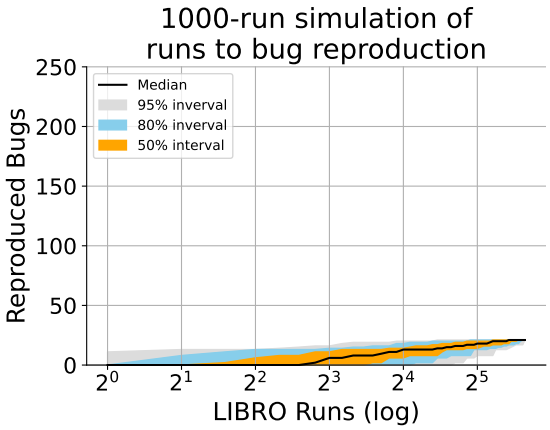


(e) Results for the test series *Experiment 4*.

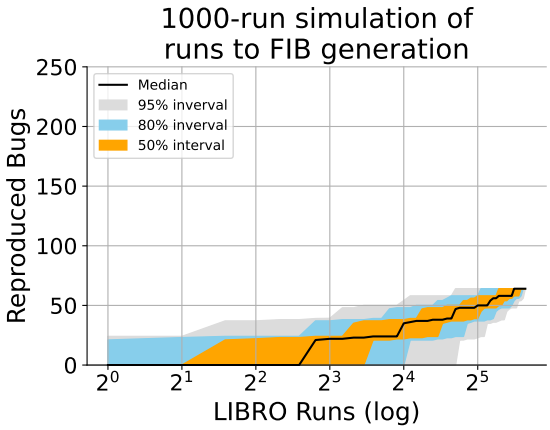


(f) Results for the test series *All Ingredients*.

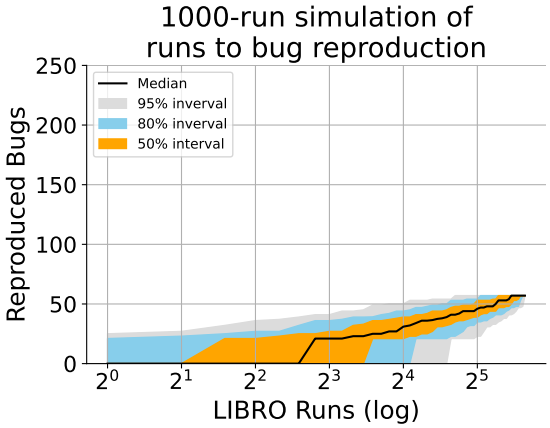
Figure 12: Distribution of the *max_output_clus_size* values for reproduced and not-reproduced bugs for the Defects4J dataset.



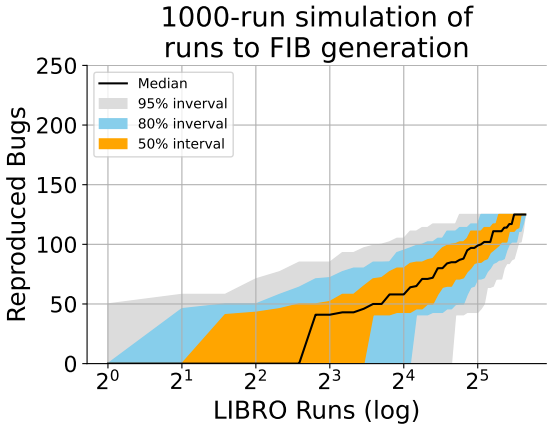
(a) Generation attempts to performance for bug reproduction for *Basic*.



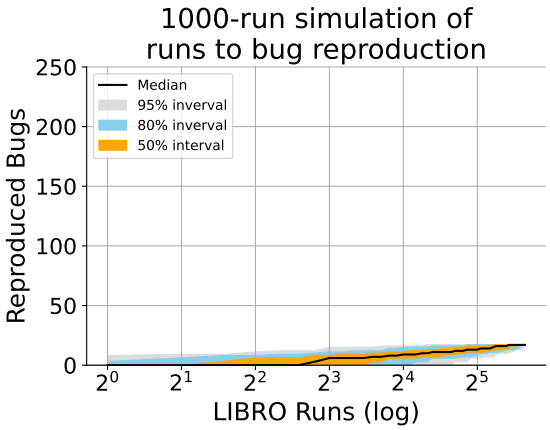
(b) Generation attempts to performance for FIB for *Basic*.



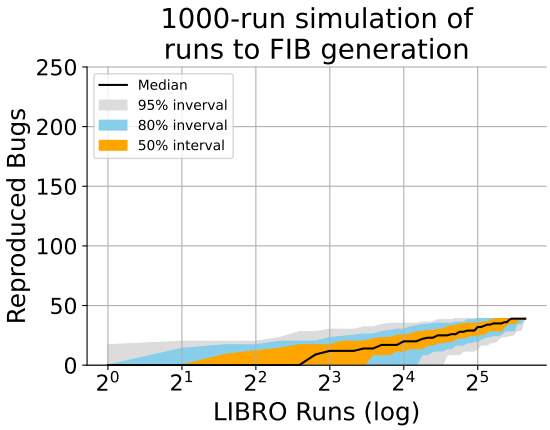
(c) Generation attempts to performance for bug reproduction for *Experiment 1*.



(d) Generation attempts to performance for FIB for *Experiment 1*.

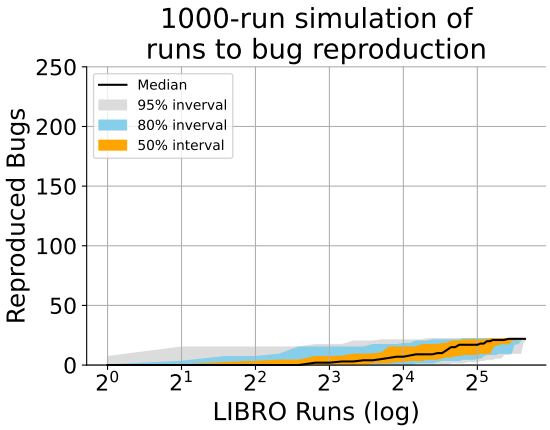


(e) Generation attempts to performance for bug reproduction for *Experiment 2*.

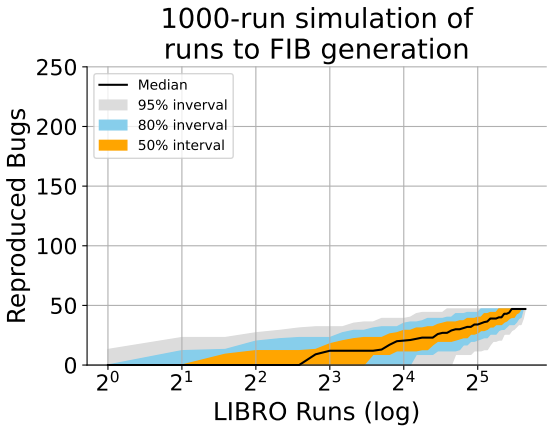


(f) Generation attempts to performance for FIB for *Experiment 2*.

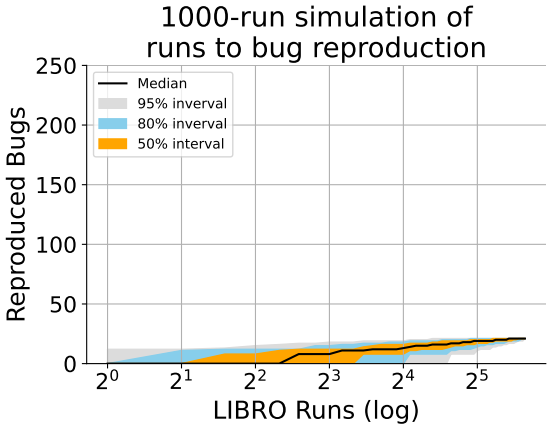
Figure 13: Generation attempts to performance for bug reproduction (left) and for FIB (right) for the Defecs4J dataset.



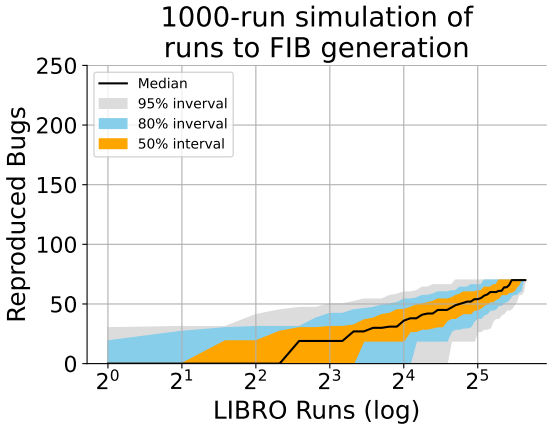
(g) Generation attempts to performance for bug reproduction for *Experiment 3*.



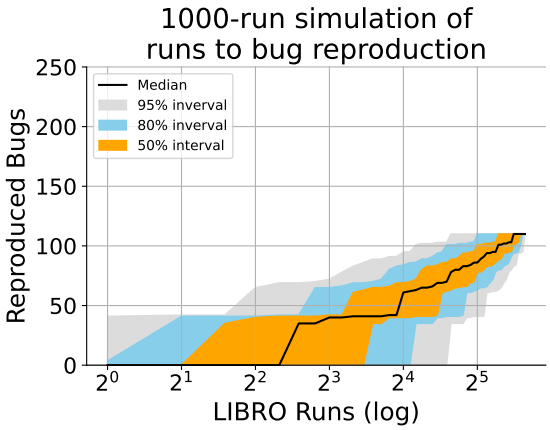
(h) Generation attempts to performance for FIB for *Experiment 3*.



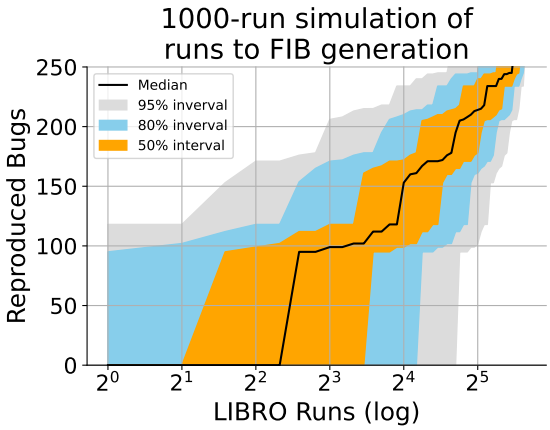
(i) Results for bug reproduction for *Experiment 4*.



(j) Generation attempts to performance for FIB for *Experiment 4*.



(k) Results for bug reproduction for *All Ingredients*.



(l) Generation attempts to performance for FIB for *All Ingredients*.

Figure 13: Generation attempts to performance for bug reproduction (left) and for FIB (right) for the Defecs4J dataset.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 22. Dezember 2025