

LLM4SZZ: Enhancing SZZ Algorithm with Context-Enhanced Assessment on Large Language Models

LINGXIAO TANG*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

JIAKUN LIU, Singapore Management University, Singapore

ZHONGXIN LIU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

XIAOHU YANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

LINGFENG BAO^{†*}, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

The SZZ algorithm is the dominant technique for identifying bug-inducing commits and serves as a foundation for many software engineering studies, such as bug prediction and static code analysis, thereby enhancing software quality and facilitating better maintenance practices. Researchers have proposed many variants to enhance the SZZ algorithm's performance since its introduction. The majority of them rely on static techniques or heuristic assumptions, making them easy to implement, but their performance improvements are often limited. Recently, a deep learning-based SZZ algorithm has been introduced to enhance the original SZZ algorithm. However, it requires complex preprocessing and is restricted to a single programming language. Additionally, while it enhances precision, it sacrifices recall. Furthermore, most of variants overlook crucial information, such as commit messages and patch context, and are limited to bug-fixing commits involving deleted lines.

The emergence of large language models (LLMs) offers an opportunity to address these drawbacks. In this study, we investigate the strengths and limitations of LLMs and propose LLM4SZZ, which employs two approaches (i.e., rank-based identification and context-enhanced identification) to handle different types of bug-fixing commits. We determine which approach to adopt based on the LLM's ability to comprehend the bug and identify whether the bug is present in a commit. The context-enhanced identification provides the LLM with more context and requires it to find the bug-inducing commit among a set of candidate commits. In rank-based identification, we ask the LLM to select buggy statements from the bug-fixing commit and rank them based on their relevance to the root cause. Experimental results show that LLM4SZZ outperforms all baselines across three datasets, improving F1-score by 6.9% to 16.0% without significantly sacrificing recall. Additionally, LLM4SZZ can identify many bug-inducing commits that the baselines fail to detect, accounting for 7.8%, 7.4% and 2.5% of the total bug-inducing commits across three datasets, respectively.

CCS Concepts: • Software and its engineering → Software maintenance tools.

Additional Key Words and Phrases: SZZ Algorithm, large language model

*Also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

[†]Corresponding author

Authors' addresses: Lingxiao Tang, 12421037@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China; Jiajun Liu, jkliu@smu.edu.sg, Singapore Management University, Singapore, Singapore; Zhongxin Liu, liu_zx@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China; Xiaohu Yang, yangxh@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China; Lingfeng Bao, lingfengbao@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

XXXX-XXXX/2025/4-ART \$15.00

<https://doi.org/XXXXXX.XXXXXXX>

ACM Reference Format:

Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. 2025. LLM4SZZ: Enhancing SZZ Algorithm with Context-Enhanced Assessment on Large Language Models. 1, 1 (April 2025), 24 pages. <https://doi.org/XXXXXX.XXXXXXXX>

1 INTRODUCTION

Having been proposed in 2005, the SZZ algorithm [50] and its variants have been widely used in finding bug-inducing commits from bug-fixing commits. The original SZZ algorithm assumes that the deleted lines in the bug-fixing commit cause the bug. It first locates the deleted lines in the bug-fixing commit. Then, it uses the annotate command from the version control system to trace back the commits that most recently added or modified these lines. Finally, it marks the identified commits as bug-inducing commits. Many downstream tasks can be performed based on bug-inducing commits, such as analyzing why the bugs occur [2, 4], predicting defects [16, 20, 68], and measuring the factors that influence software quality [9, 55].

Although the SZZ algorithm has achieved great success, it still suffers from low precision. Consequently, many variants have been proposed [12, 13, 28, 40, 52] to address this problem. Some methods [12, 28, 40] attempted to improve precision by removing noise in bug-fixing commits using static analysis. Noise refers to changes that do not influence the program's behavior, such as blank lines, comments, or refactoring operations. These irrelevant changes are unrelated to the bug, and tracing them back can lead to false positives in the output. Other methods [13] try to improve precision by treating the commits identified by the original SZZ algorithm as candidates and selecting the final bug-inducing commit from them. They choose the final bug-inducing commit by considering factors such as commit dates or the number of changed lines. To further improve precision, Tang et al. [52] introduced a deep learning method that embeds changed lines based on their semantic meanings and relationships, training a ranking model to identify the deleted lines most likely to cause the bug. However, this approach significantly sacrifices recall.

Although previous studies have made some advancements, several limitations still exist. **Limitation 1:** These methods overlook the commit message of the bug-fixing commit. Typically, the commit message contains essential information on why the changes were made [39, 67] and many of these messages describe how the bug occurs and how the commit fixes it. This information is vital for understanding the commit and accurately locating buggy statements. **Limitation 2:** These methods assume that only deleted lines cause bugs [28, 50, 52], making them inapplicable to bug-fixing commits that contain only added lines. **Limitation 3:** These methods focus solely on changed lines, ignoring the context of the entire patch. Previous studies have shown that the context, including unmodified lines near the changes, can provide crucial information for the model to understand the code [11, 61]. Sometimes, it might be the unmodified lines themselves that lead to the bug, rather than the changed lines [47]. **Limitation 4:** Methods that select the final bug-inducing commit from a set of candidates often rely on heuristic assumptions [13], such as commit dates or the number of changed lines. These assumptions may not work in all scenarios [3]. Ideally, we should determine the final bug-inducing commit based on the root cause of the bug and the content of the candidate commit.

The emergence of Large Language Models (LLMs) presents an opportunity to address the aforementioned limitations. Previous studies indicate that LLMs can effectively understand code changes and commit messages [30, 66]. One fundamental improvement is to utilize the LLM to analyze the root cause of the bug and identify buggy statements based on code changes and the commit message. This process leverages the commit message, addressing limitation 1. When identifying buggy statements, the LLM can detect not only deleted lines but also unchanged lines, addressing limitation 2. The enhanced approach then traces these buggy statements to obtain a

set of candidate commits and requires the LLM to select the bug-inducing commits from this set, addressing limitation 4. Furthermore, we can provide the LLM with more context which solves limitation 3. At first glance, this simple approach seems to address all the problems. However, several challenges remain in this simple method. **Challenge 1:** LLMs struggle with complex bug-fixing commits that involve numerous changes across multiple files and functions. These commits often contain significant noise unrelated to the bug fix, undermining LLM’s performance. **Challenge 2:** We need to provide more information to help the LLM determine whether the bug exists. The root cause of the bug and the content of the commit are often insufficient (see Section 3.2 and Section 5.2). **Challenge 3:** When asking the LLM to determine whether a commit contains a bug, we must carefully consider the context provided. An overly long context can degrade performance [32], while a too-short context may omit crucial information necessary for the LLM to understand the code [11, 61]. **Challenge 4:** Many types of bugs remain beyond the LLM’s understanding [6, 45], making it difficult for LLMs to ascertain their presence in a commit. Treating these bugs the same way as those the LLM can comprehend will adversely affect overall performance. For instance, if we determine that the LLM can understand the bug and identify its presence in a commit, we can use it to select the final bug-inducing commit from a set of candidates; otherwise, we cannot. Further details will be discussed in Section 5.2. Therefore, a better approach is needed to solve those challenges to fully leverage the potential of LLMs.

In this paper, we propose an LLM-based approach called LLM4SZZ. In the preparation phase, we summarize the root cause of the bug and filter out irrelevant files based on the patch content and the commit message. This step helps eliminate noise when handling large bug-fixing commits, addressing challenge 1. Next, we assess the LLM’s ability to understand the bug and the ability to determine whether it exists in the commit. Instead of directly asking the LLM to determine whether a commit contains a bug, we employed a more complex strategy, consisting of several parts. This approach is taken because we find that direct judgments are ineffective, see Section 5.2. To evaluate this ability, we first provide the LLM with expanded context and require it to generate a hint indicating whether the bug is present, addressing challenge 2. Before asking the LLM to determine whether the bug exists, we refine the context to address challenge 3. We then present the LLM with the root cause of the bug, the hint, and refined contexts for two versions of the program: one buggy and one correct. If the LLM can accurately distinguish between the two versions, we consider it capable; otherwise, it is not. Based on this ability assessment, we developed two approaches: context-enhanced identification and rank-based identification, which resolves challenge 4. In context-enhanced identification, we provide the LLM with more context and require it to select the bug-inducing commit from a set of candidates. In rank-based identification, we follow the methodology outlined in the previous study [52], asking the LLM to identify buggy statements from the bug-fixing commit and rank them based on their relevance to the root cause. To evaluate our method, we use three high-quality, developer-annotated datasets, ensuring their accuracy. We assess our proposed method by answering the following questions:

RQ1: How effective is LLM4SZZ in identifying bug-inducing commits from bug-fixing commits compared to baselines?

In this RQ, we compare LLM4SZZ with all baselines across three datasets to determine whether LLM4SZZ can outperform the baselines in identifying bug-inducing commits. The experimental results demonstrate that LLM4SZZ surpasses all other baselines in F1-score, with a notable improvement ranging from 6.9% to 16.0%. Furthermore, LLM4SZZ enhances both precision and F1-score without significantly sacrificing recall.

RQ2: How effective are the key components of LLM4SZZ ?

We also conduct an ablation experiment to ensure that all key components of LLM4SZZ, namely the context-enhanced assessment, the context-enhanced identification, and the rank-based identification, contribute to its performance. Additionally, we demonstrate that utilizing LLMs directly on the SZZ algorithm cannot yield satisfactory results.

RQ3: How effective is LLM4SZZ if we apply it on other open-source large language models?

In this RQ, we aim to examine whether the core ideas of LLM4SZZ can be applied to other open-source large language models. We implement LLM4SZZ using llama3-8b and llama3-70b. The experimental results show that LLM4SZZ can be effectively applied to other LLMs, and better LLMs can enhance its performance.

In summary, we make the following contributions:

- We provide insights into how large language models (LLMs) can enhance the performance of the SZZ algorithm while also highlighting the limitations in this task.
- Based on these insights, we propose a novel approach to fully leverage the LLM’s capabilities, which consists of two methods for locating bug-inducing commits, with the choice of method being adaptive to the LLM’s ability to comprehend the bug.
- We implement LLM4SZZ on two popular programming languages and evaluate it on three developer-annotated datasets. The experimental results show that LLM4SZZ outperforms all other baselines across the datasets.

2 BACKGROUND

In this section, we first introduce the SZZ algorithm’s variants. Then we present our motivation examples.

2.1 SZZ algorithms

AG-SZZ. The AG-SZZ algorithm was proposed by Kim et al. [28]. They observed that some changes in bug-fixing commits, such as blank lines, comments, and cosmetic changes, do not affect the program’s behavior. Therefore, they excluded these changes when tracing back deleted lines. Additionally, they utilized the annotation graph instead of simply using the annotate command, as the annotation graph provides more detailed information about line changes and movements.

MA-SZZ. Da Costa et al. proposed the MA-SZZ algorithm. [12]. They found that the AG-SZZ algorithm mistakenly identifies commits with only meta-changes as bug-inducing commits. Meta-changes refer to branch changes, merge changes, and property changes. Da Costa et al. addressed this issue by connecting all meta-change nodes in the annotation graph to their prior changes, ensuring that the MA-SZZ algorithm does not include meta-changes as bug-inducing commits.

R-SZZ and L-SZZ. L-SZZ and R-SZZ, both based on the AG-SZZ algorithm, were proposed by Davies et al. [13]. They improved the AG-SZZ algorithm by selecting only one commit as the bug-inducing commit from the results produced by the AG-SZZ algorithm. R-SZZ selects the commit with the most recent date, while L-SZZ selects the commit with the most changed lines.

RA-SZZ. Neto et al. [40] proposed the RA-SZZ algorithm after discovering that previous SZZ algorithms trace back changed lines related to refactoring operations when locating bug-inducing commits. Since refactoring operations do not affect the program’s behavior, including them may introduce noise. Therefore, they used two tools RefDiff [49] and Refactoring Miner [54] to exclude refactoring modifications before tracing back lines. However, this algorithm is limited to Java projects, as the two tools mentioned above cannot work on other programming languages.

Neural-SZZ. Neural-SZZ, proposed by Tang et al. [52], is based on deep learning. They observed that the previous methods fail to consider the semantic meaning of changed lines and the relationships between them. To address this, they utilize the CodeBERT [18] model to embed the changed lines,

Fixing Commit: eed6e41813d in linux

driver core: Fix locking bug in deferred_probe_timeout_work_func(). list_for_each_entry_safe() is only useful if we are deleting nodes in a linked list within the loop. It doesn't protect against other threads adding/deleting nodes to the list in parallel.

1 files changed, 5 additions(+) and 3 deletions(-)

dd.c

```

1 static void deferred_probe_timeout_work_func(struct work_struct *work)
2 {
3 -     struct device_private *private, *p;
4 +     struct device_private *p;
5     driver_deferred_probe_timeout = 0;
6     driver_deferred_probe_trigger();
7     flush_work(&deferred_probe_work);
8 -     list_for_each_entry_safe(private, p, &deferred_probe_pending_list, deferred_probe)
9 -         dev_info(private->device, "deferred probe pending\n");
10+    mutex_lock(&deferred_probe_mutex);
11+    list_for_each_entry(p, &deferred_probe_pending_list, deferred_probe)
12+        dev_info(p->device, "deferred probe pending\n");
13+    mutex_unlock(&deferred_probe_mutex);
14+    wake_up_all(&probe_timeout_waitqueue);
15 }
16 static DECLARE_DELAYED_WORK(deferred_probe_timeout_work,
17     deferred_probe_timeout_work_func);

```

Fig. 1. Motivation example one

capturing their semantic meanings. Additionally, they use a heterogeneous graph attention network (HAN) [56] to capture the relationships between changed lines. After obtaining the embeddings of the changed lines, they employ the RankNet [7] model to select the deleted lines that are most likely to be the root cause of the bug. Finally, they trace back the top N lines in the ranked list to locate bug-inducing commits. The authors implemented the algorithm only for the Java programming language.

2.2 Potential and limitations of LLMs

In this subsection, we present motivation examples to demonstrate the potential and limitations of LLMs in locating bug-inducing commits. We utilize the LLM GPT-4o-mini [43] to illustrate these examples.

LLMs have the potential to identify the root cause of the bug from the bug-fixing commit and reduce false positives by pinpointing the bug-inducing commit from a set of candidates. We illustrate this with the example presented in Figure 1, which involves a bug-fixing commit *eed6e41813d* in Linux. We feed the prompt, "Based on the content of the bug-fixing commit, analyze the root cause of the bug and output the code statements leading to the bug", along with the content of the bug-fixing commit to the LLM. The LLM successfully predicts that the bug occurs because the function *list_for_each_entry_safe* fails to protect the list when multiple threads add or delete nodes in parallel. It identifies lines 8 and 9 as buggy statements, filtering out line 3. Tracing back these two lines will yield two candidate bug-inducing commits, *eb7fbc9fb11* and *25b4e70dcce*. We then use the LLM to determine which candidate commit introduces the bug. The LLM finds that the commit *eb7fbc9fb11* introduces line 9 but only modifies the second parameter of the *dev_info* function, which does not affect the existence of the bug. Consequently, we filter out commit *eb7fbc9fb11* and identify *25b4e70dcce* as the final bug-inducing commit.

However, LLMs face challenges when handling large bug-fixing commits, so it is beneficial to filter out irrelevant files before identifying buggy statements. This is demonstrated in the second motivation example illustrated in Figure 2. This commit modifies four files, introducing twenty-nine insertions and making eight deletions. Due to page limits, we only show the most

Fixing Commit: c5153331c in Accumulo

Enforce a valid instance name on ZKI creation by calling getInstanceID(), which would throw a RuntimeException if the user passed in an instance name which did not exist in the zookeepers provided.....

4 files changed, 29 insertions(+), 8 deletions(-)

ZooKeeperInstance.java

```

1 ZooKeeperInstance(Configuration config, ZooCacheFactory zcf) {
2     ArgumentChecker.notNull(config);
3     if (config instanceof ClientConfiguration) {
4         this.clientConf = (ClientConfiguration) config;
5     } else {
6         this.clientConf = new ClientConfiguration(config);
7     }
8     this.instanceId = clientConf.get(ClientProperty.INSTANCE_ID);
9     this.instanceName = clientConf.get(ClientProperty.INSTANCE_NAME);
10    if ((instanceId == null) == (instanceName == null))
11        throw new IllegalArgumentException("Expected exactly one of instanceName and instanceId to be set");
12    this.zooKeepers = clientConf.get(ClientProperty.INSTANCE_ZK_HOST);
13    this.zooKeepersSessionTimeout = (int)AccumuloConfiguration.getTimeInMillis(clientConf.get(ClientProperty.INSTANCE_ZK_TIMEOUT));
14    zooCache = zcf.getZooCache(zooKeepers, zooKeepersSessionTimeout);
15+   if (null != instanceName) {
16+     // Validates that the provided instanceName actually exists
17+     getinstanceID();
18+   }
19 }
```

Fig. 2. Motivation example two

important part related to the identification of bug-inducing commits. The lines highlighted in blue are added by us and are not in the original patch content. According to the commit message, the bug arises because the program fails to call the getInstanceId function to enforce a valid instance name and the bug is only related to the instanceName variable. If we directly input the whole patch into the LLM and require it to identify the code statements leading to the bug, it erroneously points the @Test(expected = RuntimeException.class) statement in another file named ZooKeeperInstanceTest.java.

This example also demonstrates that LLMs have the potential to understand commit messages and accurately locate buggy statements, but they need sufficient context. If we exclude the other files and only feed the LLM with changes in the correct file ZooKeeperInstance.java, the LLM still cannot output the correct code statements. Concretely, if we provide the LLM with the commit message and the original patch content (lines 12 to 19 in Figure 2), it still incorrectly identifies line 14 as buggy code statements. This is due to insufficient context. According to the commit message, the bug is related to the variable instanceName. However, in the original patch content (lines 12 to 19), the only code statement related to the variable instanceName is line 15, which is used to fix the bug. The full content of the ZooKeeperInstance constructor (lines 1 to 19 in Figure 2) contains the statement this.instanceName = clientConf.get(ClientProperty.INSTANCE_NAME), which relates to the instanceName variable. But this statement is not displayed in the original patch. By providing the expanded context (lines 1 to 19), which includes the entire constructor, the LLM can correctly identify the code statement in line 9.

3 APPROACH

Building on the motivation examples, we propose a new framework called LLM4SZZ to effectively detect buggy statements and locate bug-inducing commits. Fig. 3 presents the overview of our framework, which consists of three parts: preparation, context-enhanced assessment, and commits identification. In the preparation phase, we analyze the bug-fixing commit, identify the core files related to the bug, and determine its root cause. During the context-enhanced assessment, we assess whether the LLM can understand the bug and determine its presence in the commit. If the

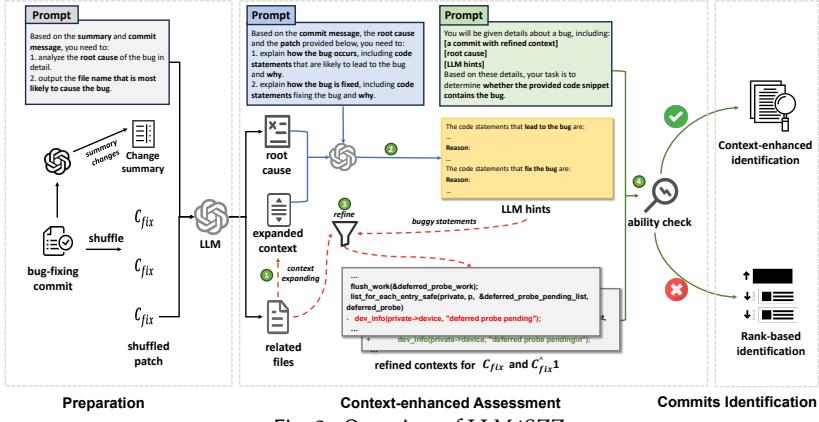


Fig. 3. Overview of LLM4SZZ

LLM demonstrates this ability, we employ the context-enhanced identification approach during the commit identification process; otherwise, we fall back to the rank-based identification approach.

3.1 Preparation

In this step, we use the large language model (LLM) to analyze bug-fixing commits. We aim to summarize the root cause of the bug based on the bug-fixing commit and filter out irrelevant files. In motivation example one, we have shown that irrelevant files undermine the LLM’s ability and we need to filter them out.

Following the chain-of-thought (CoT) concept [58], we first require the LLM to analyze the patch, summarizing the modifications and their interrelationships within the bug-fixing commit. Next, we ask the LLM to identify the root cause of the bug and the related files based on the modification summary and the commit message.

To enhance performance when handling large bug-fixing commits with multiple modified files, we employ two additional approaches. First, we shuffle the sequence of modified files in the patch, ensuring that each file has an equal chance of being identified as related to the root cause. Previous studies [35] reveal that LLMs tend to ignore content in the middle of text when handling long texts. Second, we run the LLM three times for the same question and shuffle the patch at each run-time. This approach is similar to a voting system [57]. However, instead of only considering files with majority votes, we take a more conservative approach: if a file name appears in any of the LLM outputs, we regard it as related to the root cause. This strategy helps minimize the risk of omitting important files. After this step, we obtain the root cause of the bug and filter out all irrelevant files.

3.2 Context-enhanced assessment

In this section, we explain the necessity of the assessment and our approach to it. In the first motivation example, we demonstrate that LLMs can determine whether a bug exists in a commit, allowing us to use them to select the final bug-inducing commit from a set of candidates. The second example illustrates that providing more context can enhance the LLM’s ability to understand the patch and help identify buggy statements more accurately. However, previous studies in automatic program repair have shown that LLMs still do not comprehend certain bugs [6, 45], even with enough context. This indicates that LLMs are unable to determine whether these kinds of bugs exist in programs because they cannot understand the bugs. If LLMs cannot understand the bug even with additional context, providing more context becomes meaningless, and alternative methods are required to address these cases. Therefore, it is crucial to assess the LLM’s ability to comprehend the bug and identify its presence.

To assess the LLM’s ability to determine whether the bug exists, we need two versions of the program: one containing the bug and another where it has been fixed. Therefore, an ideal approach is to make use of the bug-fixing commit C_{fix} , where the version $C_{fix}^{\wedge}1$ is buggy and version C_{fix} is correct. Although we can directly require the LLM to assess whether the commit is buggy based on the root cause of the bug, experimental results indicate that this approach yields low performance (see section 4). Instead, we first require the LLM to generate a hint to assist in determining whether the commit contains the bug. The hint includes detailed information about the code statements in the patch. Its further specifics will be provided later in this section. Then, we separately feed the hint and the bug-related contexts extracted from commit C_{fix} and $C_{fix}^{\wedge}1$ to the LLM, asking it to identify whether each version contains the bug. If the LLM even cannot identify the two versions correctly using its own produced hint, we regard that the LLM is unable to comprehend the bug, let alone select the final bug-inducing commit from a set of candidates.

As shown in Figure 3, the entire context-enhanced assessment process consists of four steps, which are as follows:

① Context Expanding: First, we provide the LLM with sufficient context through a process that we call context expanding. Previous studies have found that the contextual code is crucial for providing information to the model [11, 61]. However, the bug-fixing commit often does not contain the full content of the changed functions. The partial content of functions in the fixing commit may hinder the LLM’s ability to understand both the functions and the modifications. Therefore, for each modified function, we expand its context by extracting the full content of both buggy and fixed versions and generating their diffs. For modified lines outside the function, we expand their context by extracting three unmodified lines around them. We have presented an example of context expanding in the second motivation example, as illustrated in Figure 2. Specifically, the code highlighted in blue represents new additions, and the others are collected from the original patch. As indicated in Section 2.2, the buggy code `this.instanceName = clientConf.get(ClientProperty.INSTANCE_NAME)` is not located in the original patch but in the expanded context, indicating the necessity of the context expanding.

② Hint Generation: Next, we require the LLM to establish a hint to determine whether the bug exists in a commit. Specifically, we ask the LLM to identify the code statements leading to the bug and provide a reason. Additionally, we require the LLM to identify the code statements that fix the bug and provide a reason. Note that we do not limit the LLM to choosing code statements only from deleted lines. It can select any code statements from the expanded context.

③ Context Refinement: Before assessing the LLMs’ ability to determine whether the bug exists in a commit, we need to refine the expanded context to obtain the refined context. This step is necessary because the expanded context may contain much irrelevant content that is not related to the bug. For example, the expanded context might include an entire function with hundreds of lines, while only a few lines are relevant to the bug. Feeding the expanded context directly to LLMs may undermine their ability to assess the existence of the bug in a commit, as previous studies [32] suggest that LLMs struggle with intricate tasks when handling long texts. Therefore, we attempt to refine the expanded context. We first extract the buggy statements identified in the hint from the file in commit $C_{fix}^{\wedge}1$. These buggy statements are then sorted in ascending order based on their line numbers $\{l_1, l_2, \dots, l_n\}$ in commit $C_{fix}^{\wedge}1$, where l_1 is the smallest line number and l_n is the largest. Here, we define l_{min} as $l_1 - N$ and l_{max} as $l_n + N$. N is a constant starting from 3 to ensure that lines l_{min} and l_{max} can be mapped to corresponding lines in commit C_{fix} . If line l_{min} or line l_{max} cannot be mapped, we keep incrementing N . Then, we extract the content ranging from the line number l_{min} to the line number l_{max} , forming the refined context for commit $C_{fix}^{\wedge}1$. To obtain the refined context for commit C_{fix} , we map lines l_{min} and l_{max} to their corresponding line

The code statements that lead to the bug are:
`list_for_each_entry_safe(private, p, &deferred_probe_pending_list, deferred_probe)`
`dev_info(private->device, "deferred probe pending\n");`

Reason: The provided code snippet is indeed buggy because it uses 'list_for_each_entry_safe()' to traverse the 'deferred_probe_pending_list' without protecting against concurrent modifications from other threads...

The code statements that fix the bug are:
`mutex_lock(&deferred_probe_mutex);`
`mutex_unlock(&deferred_probe_mutex);`

Reason: The provided code snippet correctly uses a mutex lock ('mutex_lock(&deferred_probe_mutex);') before iterating over the 'deferred_probe_pending_list' with 'list_for_each_entry()...'.

Hint

```

1 driver_deferred_probe_timeout = 0;
2 driver_deferred_probe_trigger();
3 flush_work(&deferred_probe_work);
4- list_for_each_entry_safe(private, p, &deferred_probe_pending_list,
5-     deferred_probe)
6- dev_info(private->device, "deferred probe pending\n");
7 }
8 static DECLARE_DELAYED_WORK(deferred_probe_timeout_work,
9     deferred_probe_timeout_work_func);

```

```

1 driver_deferred_probe_timeout = 0;
2 driver_deferred_probe_trigger();
3 flush_work(&deferred_probe_work);
4+ mutex_lock(&deferred_probe_mutex);
5+ list_for_each_entry(p, &deferred_probe_pending_list,
6     deferred_probe)
7+ dev_info(p->device, "deferred probe pending\n");
8+ mutex_unlock(&deferred_probe_mutex);
9+ wake_up_all(&probe_timeout_workqueue);
10 }
11 static DECLARE_DELAYED_WORK(deferred_probe_timeout_work,
12     deferred_probe_timeout_work_func);

```

eed6e41813d¹
eed6e41813d

Fig. 4. An example of context-enhanced ability check

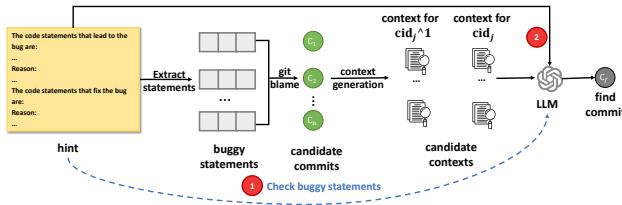


Fig. 5. The workflow of context-enhanced identification

numbers l'_{min} and l'_{max} and extract the content between these two line numbers in commit C_{fix} , forming the refined context for commit C_{fix} .

④ Ability Check: Finally, with the contexts for both commits obtained, we begin to check the LLM's ability to determine whether the bug exists in the commit. We provide the LLM with the root cause of the bug, the hint collected above, and the refined contexts for two versions. If the LLM can correctly identify the fixed version and the buggy version, we proceed to adopt context-enhanced identification. Otherwise, we fall back to rank-based identification.

One example of the context-enhanced ability check is shown in Figure 4, which corresponds to the motivation example one. Here, we set N to 3. The LLM identifies lines from 4 to 6 as buggy statements in $eed6e41813d^1$. Therefore, l'_{min} is 1 and l'_{max} is 9. To generate the refined context for this commit, we extract lines ranging from line 1 to line 9. We then map line 1 and line 9 to commit $eed6e41813d$, getting l'_{min} as 1 and l'_{max} as 12. Finally, we extract the lines between them, forming the refined context in commit $eed6e41813d$. We feed the LLM with the root cause of the bug, the hint, and two versions of the context. It identifies the context for commit $eed6e41813d^1$ as buggy and the context for commit $eed6e41813d$ as correct, demonstrating its ability to determine whether the bug exists in a commit.

3.3 Commits identification

3.3.1 Context-enhanced identification. Given a bug-fixing commit, once we verify that the LLM can understand the bug in it with the expanded and refined context, we apply **context-enhanced identification** to identify the bug-inducing commits, as shown in Figure 5. First, we retrieve all

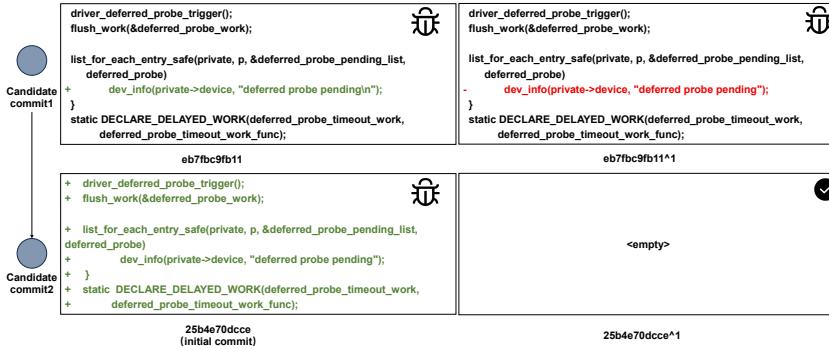


Fig. 6. An example of context-enhanced identification

buggy statements from the hint. We then trace back these buggy code statements, obtain a set of candidate commits, and sort them in descending order by commit date, forming a candidate list $\{C_1, C_2, \dots, C_i\}$. Next, we generate the refined context for each candidate commit following the same process used in the ability check. For each candidate commit C_j , we create the context for both C_j and its previous version $C_j^{\wedge}1$.

To enhance the LLM's ability to determine whether the candidate commit C_j contains the bug, we split the determination process into two steps. In the first step, we input the context of commit C_j and the buggy statements in the hint to the LLM, asking the LLM to determine whether the commit contains the buggy code statements or code statements with similar semantic meanings to the buggy code statements. If the LLM answers "no", we simply believe that the commit C_j does not contain the bug. If the LLM answers "yes", we proceed to let it determine whether the commit is buggy. In the second step, we feed the context of commit C_j , the root cause of the bug, and the hint to the LLM, asking it to determine whether the commit contains the bug.

We utilize the LLM to check the candidate commits in the list from index 1 to i . If we can find an index f , where the LLM believes that the context of the commit C_f is buggy but $C_f^{\wedge}1$ is not, we designate it as the bug-inducing commit. If we can not find such a commit, we fall back to rank-based identification conservatively.

Figure 6 provides an example, which corresponds to the first motivation example. Firstly, LLM4SZZ traces back the buggy statements in the hint and finds two candidate commits, the commit $eb7fbc9fb11$ (denoted as C_1) and commit $25b4e70dcce$ (denoted as C_2). We sort them based on their commit date in descending order and get the candidates list $\{C_1, C_2\}$. We first check the contexts for commits C_1 and $C_1^{\wedge}1$, following the steps above and requiring the LLM to determine whether the two versions are buggy. The LLM identifies that both the C_1 and $C_1^{\wedge}1$ versions of the program contain the bug. Therefore, this candidate commit is not the bug-inducing commit. LLM4SZZ then checks the commit C_2 and $C_2^{\wedge}1$, following the same steps. Here, C_2 is the initial commit that introduces the file. Therefore, the context of $C_2^{\wedge}1$ is empty. The LLM finds the commit C_2 contains the bug and the context of commit $C_2^{\wedge}1$ contains no code statements and is not buggy. Therefore, LLM4SZZ finally designates the commit C_2 as the final bug-inducing commit.

3.3.2 Rank-based identification. **Rank-based identification** addresses cases where the LLM cannot fully understand the bug and cannot determine whether the bug exists in the program. Therefore, in this approach, we do not use LLMs to select the final bug-inducing commit from candidate commits. Instead, we simply follow the idea of NerialSZZ [52] and proceed as follows:

Table 1. The statistics of the bugs and corresponding bug fixing commits in three datasets

Dataset	Project	#Bug-Fixing	#Bug-Inducing	#SMALL	#LARGE
DS_LINUX	linux	1,500	1,562	681	819
	systemd	15	15	5	10
	qemu	10	10	3	7
	gpac	9	9	2	7
	unitime	6	6	2	4
	JohnTheRipper	5	5	2	3
	libvirt	4	4	2	2
	opensips	4	4	1	3
	... (279 more projects)	308	309	130	171
Total		361	362	146	207
DS_GITHUB	accumulo	35	55	7	28
	ambari	38	44	1	37
	hadoop	53	57	6	47
	lucene	70	145	3	68
	oozie	45	50	3	42
	Total	241	351	20	222

- **Buggy Statements Identification:** We first ask the LLM to identify buggy statements from the bug-fixing commit based on the commit message and the root cause obtained in section 3.1. In this phase, we input only the root cause, the commit message, and the original changed files obtained in section 3.1. We do not provide the LLM with expanded context, as we observe that if the LLM cannot understand the bug, additional context will undermine its performance (see Section 5.2).
- **Relevance Ranking:** By utilizing a listwise rank algorithm [51] based on the LLM, we rank these buggy statements according to their relevance to the root cause.
- **Candidate Commits generation:** For each file, we retrieve the top N code statements, trace them back to their corresponding commits, and add these commits to our list of candidate commits.
- **Final Commit Designation:** We then sort these candidate commits by their commit date and designate the most recent commit as the bug-inducing commit. This approach aligns with previous studies [3, 46], which suggest that bugs are typically introduced by recent commits.

4 EXPERIMENT SETUP

4.1 Dataset

To evaluate our method, we require high-quality datasets containing bug-fixing commits and their corresponding bug-inducing commits. Previous research [59] has demonstrated that datasets produced by the SZZ algorithm contain significant noise, leading us to discard these datasets. Other available datasets are annotated by researchers [13, 41]. While these datasets are of higher quality, researchers may not have the same level of knowledge as developers about specific projects, which still can result in inaccuracies. To ensure accuracy, we combined three developer-annotated datasets to form the final dataset for evaluating our method. In these datasets, all bug-fixing commits and bug-inducing commits are annotated by developers, and are extracted from bug reports or commit messages.

DS_LINUX refers to the dataset created by Lyu et al. [36], which is based on the Linux kernel. The researchers observed that Linux developers label bug-fixing commits with their corresponding bug-inducing commits in the commit messages. They collected these commit messages and built the dataset based on them. This dataset is notable for its size, containing 76,046 pairs of bug-fixing and bug-inducing commits. However, its drawback is that it is only related to the Linux kernel.

DS_GITHUB refers to the dataset constructed from multiple repositories on GitHub, collected by Rosa et al. [47]. The authors mined GitHub by first locating the bug-fixing commits and then

identifying the corresponding bug-inducing commits based on the information left by developers in the commit messages. This dataset is characterized by its inclusion of hundreds of repositories. However, its drawback is that each repository contains very few bug-fixing commits. Moreover, this dataset includes many repositories with few stars.

DS_APACHE refers to the dataset created from several Apache projects, collected by Wen et al. [59]. The researchers extracted the bug reports from several Apache projects, obtaining bug-fixing commits and their corresponding bug-inducing commits based on the bug reports and commit messages. This dataset contains several Apache projects with high star ratings, and each project corresponds to a moderate number of bug-fixing commits.

Table 1 presents the statistics of the three datasets. Since the majority of the combined datasets are comprised of C and Java projects, we include only C and Java projects in the final dataset. To control experimental costs, we sample data from DS_LINUX following previous studies [31, 66, 69], using a 95% confidence level and a margin of error below 5%. We sample 1,500 bug-fixing commits from a total of 76,046 commits, along with their corresponding bug-inducing commits. This sample size is comparable to previous studies. For example, Li et al. [31] sampled 381 commits from a total of 35,431 commits to evaluate their approach for generating commit messages. Note that we do not sample data from DS_GITHUB and DS_APACHE. We also provide information about the size of bug-fixing commits. Following previous studies [3, 52], if a bug-fixing commit contains more than five changed lines, we categorize it as a large commit, otherwise, we categorize it as a small commit. From the table, we observe that the number of small bug-fixing commits is roughly equal to the number of large bug-fixing commits in DS_LINUX and DS_GITHUB, while most bug-fixing commits in DS_APACHE are large. In summary, our dataset comprises multiple high-quality pairs of bug-fixing and bug-inducing commits in various programming languages across numerous repositories.

4.2 Experiment Setting

Our experiment is conducted on a server equipped with two NVIDIA A800 GPUs and an Intel Xeon 6326 CPU, running on Ubuntu OS. We utilize gitpython [19] to extract patch content and obtain the necessary information about commits. Additionally, we use the *tree-sitter* [53] parser to extract functions from the source code when generating the context. Although our implementation focuses on the Java and C programming languages, LLM4SZZ is generic and can be easily extended to other programming languages by altering the parser in *tree-sitter*.

For LLMs, we aim to balance cost and effectiveness for proprietary models. Therefore, we choose GPT-4o-mini due to its low fees and relatively high effectiveness. We estimated that using GPT-4 would cost approximately \$300 per round, which is prohibitively expensive. Our experiments show that GPT-4o-mini is sufficient for our needs. For open-source LLMs, we use Llama3-8b and Llama3-70b, which we downloaded from Hugging Face [23]. To reduce randomness, we set the temperature to 0.0 for both GPT-4o-mini and the open-source LLMs, aligning with settings used in previous studies [31, 65]. We also employ two strategies to further address randomness. First, we repeat the entire experiment three times and calculate the average metrics across the three runs. Second, our dataset consists of 2,102 test cases, which is large enough to reveal statistical patterns and minimize the influence of individual test cases on the final results.

We use the SZZ algorithms introduced in Section 2.1 as our baselines. Implementations of the B-SZZ, AG-SZZ, MA-SZZ, L-SZZ, R-SZZ, and RA-SZZ algorithms are from the replication package provided by Rosa et al. [47]. The implementation of the NeuralSZZ algorithm is from the replication package provided by Tang et al. [52]. We train the model using the same training set as in the original paper and achieve nearly identical performance on its own test set. We then apply the trained model to DS_GITHUB and DS_APACHE-j. To evaluate our approach and baselines, we

Table 2. The performance comparisions between all methods in finding c bug-inducing commits

Method	DS_LINUX			DS_GITHUB-c		
	Precision	Recall	F1-score	Precision	Recall	F1-score
B-SZZ	0.452	0.578	<u>0.507</u>	0.361	0.656	0.466
AG-SZZ	0.448	0.553	0.495	0.410	0.592	0.484
MA-SZZ	0.421	0.538	0.472	0.335	0.624	0.436
R-SZZ	0.583	0.448	<u>0.507</u>	0.671	0.582	<u>0.620</u>
L-SZZ	0.560	0.430	0.486	0.486	0.422	0.452
LLM4SZZ	0.628	0.552	0.588	0.687	0.641	0.663

Table 3. The performance comparison between methods in finding java bug-inducing commits

Method	DS_GITHUB-j			DS_APACHE		
	Precision	Recall	F1-score	Precision	Recall	F1-score
B-SZZ	0.285	0.680	0.401	0.251	0.435	0.318
AG-SZZ	0.421	0.533	0.470	0.328	0.310	0.318
MA-SZZ	0.239	0.560	0.335	0.307	0.345	0.329
R-SZZ	0.538	0.467	0.500	0.497	0.288	0.364
L-SZZ	0.492	0.427	0.457	0.366	0.211	0.267
RA-SZZ	0.337	0.440	0.382	0.264	0.325	0.293
Neural-SZZ	0.556	0.486	<u>0.520</u>	0.563	0.364	<u>0.442</u>
LLM4SZZ	0.607	0.569	0.587	0.610	0.398	0.482

employ three widely used metrics: Precision, Recall, and F1-score, following the methodology used in previous studies [3, 36].

5 EXPERIMENT RESULTS

In this section, we first demonstrate the effectiveness of LLM4SZZ (RQ1). Next, we evaluate the impact of its key components (RQ2). Finally, we show that LLM4SZZ can be applied to other large language models (RQ3).

5.1 RQ1. Effectiveness of LLM4SZZ in identifying bug-inducing commits

Table 2 and 3 present the results of LLM4SZZ and baselines in identifying bug-inducing commits in C and Java projects, respectively. Since DS_GITHUB consists of both C projects and Java projects, we split it into DS_GITHUB-c and DS_GITHUB-j, containing C projects and Java projects, respectively.

As shown in Table 2, for DS_LINUX, all baselines perform almost the same, mirroring the experimental results in the original whole dataset [36]. This suggests that our selected dataset has the same statistical patterns as the original dataset. The B-SZZ and R-SZZ algorithms achieve the highest F1-scores among all baselines. Specifically, B-SZZ achieves the highest recall, while R-SZZ achieves the highest precision. We also analyze why B-SZZ outperforms its variants, such as AG-SZZ, MA-SZZ, and L-SZZ in DS_LINUX. B-SZZ outperforms AG-SZZ because Linux contains numerous bug-fixing commits related only to comments and configurations. While B-SZZ successfully identifies these commits, AG-SZZ filters them out. Similarly, MA-SZZ assumes that commits with only meta-changes are not bug-inducing, but DS_LINUX shows that developers do label such commits as bug-inducing. L-SZZ, on the other hand, assumes the commit with the most changed lines among those identified by AG-SZZ is the bug-inducing commit. However, this assumption also often fails, as shown by the dataset. These limitations explain why these variants perform worse than the B-SZZ algorithm. In DS_GITHUB-c, the R-SZZ algorithm performs the best, significantly outperforming all other baselines, with an F1-score of 0.620. LLM4SZZ achieves the highest precision and F1-score across these two datasets. In DS_LINUX, it improves precision by 7.7% and F1-score by 16.0% compared to the best baseline. In DS_GITHUB-c, it improves precision by 2.4% and F1-score by 6.9%, respectively.

Table 4. The performance comparisons in ablation study

Model	DS_LINUX			DS_GITHUB			DS_APACHE		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
LLM4SZZ-raw	0.470	0.609	0.531	0.441	0.659	0.528	0.402	0.393	0.397
LLM4SZZ-r	0.621	0.520	0.566	0.669	0.609	0.637	0.599	0.382	0.467
LLM4SZZ-re	0.560	0.511	0.534	0.633	0.567	0.598	0.584	0.383	0.463
LLM4SZZ-c	0.668	0.450	0.538	0.691	0.498	0.579	0.644	0.316	0.424
LLM4SZZ-h	0.680	0.379	0.487	0.564	0.307	0.397	0.523	0.195	0.284
LLM4SZZ	0.628	0.552	0.588	0.671	0.626	0.647	0.610	0.398	0.482

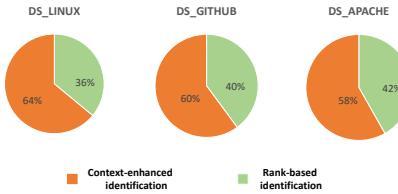


Fig. 7. The proportions of the two identification approaches in LLM4SZZ

From Table 3, we observe that the NeuralSZZ algorithm performs the best in precision and F1-score among all baselines. This suggests that utilizing deep learning to rank code statements is an effective way to enhance performance. LLM4SZZ also performs the best in precision and F1-score in these two datasets. Specifically, it improves precision by 9.2% in DS_GITHUB-j and 8.3% in DS_APACHE. Additionally, it enhances the F1-score by 12.9% in DS_GITHUB-j and 9.0% in DS_APACHE. This demonstrates the effectiveness of LLM4SZZ in handling large bug-fixing commits, as most bug-fixing commits in DS_APACHE are large.

Combining the three datasets, we observe that the R-SZZ algorithm performs better than other baselines except the Neural-SZZ algorithm. This is consistent with the finding of Rodriguez et al. [46] that defects are typically introduced in the most recent changes. Moreover, we can find that all baselines' performance varies a lot in different datasets. For example, R-SZZ outperforms other baselines a lot in DS_GITHUB-c but it performs almost the same as B-SZZ in DS_LINUX. Neural-SZZ, the deep-learning based approach, also has the same problem. It outperforms other baselines a lot in DS_APACHE but only shows a slight improvement over R-SZZ in DS_GITHUB-j. In contrast, LLM4SZZ does not have the same problem. In the worst scenario, it can still outperform other baselines by 6.9% in F1-score. Additionally, our method improves precision and F1-score without sacrificing too much recall. In all three datasets, LLM4SZZ achieves a higher recall than all other baselines, except for the B-SZZ algorithm.

RQ-1: LLM4SZZ is more precise in identifying bug-inducing commits compared to all baselines, with an increase in precision from 2.4% to 9.2%. Additionally, LLM4SZZ achieves a significant enhancement in F1-score, increasing by 6.9% to 16.0% compared to the best baselines. It also exhibits more consistent performance across the three datasets. Furthermore, LLM4SZZ improves both precision and F1-score without a substantial sacrifice in recall.

5.2 RQ2. Effectiveness of key components in LLM4SZZ

In this section, we investigate the effectiveness of key components in LLM4SZZ. In LLM4SZZ-raw, we implement the most basic setting. First, we use the LLM to analyze the root cause of the bug, and then locate the buggy statements based on this root cause. Finally, we trace back the buggy statements to identify the commits that introduced them, marking these as bug-inducing commits. In LLM4SZZ-r, we remove context-enhanced assessment and apply rank-based identification across

all test cases. The LLM4SZZ-re variant is built on LLM4SZZ-r by providing the LLM with expanded context during rank-based identification, rather than the original patch. In LLM4SZZ-c, we similarly exclude context-enhanced assessment but utilize context-enhanced identification in all scenarios. The LLM4SZZ-h variant is based on LLM4SZZ-c. It omits the hint and provides only the context of the commit along with the root cause of the bug when asking the LLM to determine whether a given commit contains the bug. This allows us to evaluate the contribution of the hint to the LLM's ability to identify the presence of the bug.

Table 4 presents the performance of the LLM4SZZ and its variants in identifying bug-inducing commits. The best results are highlighted in bold. As shown in the table, the LLM4SZZ-raw variant does not perform very well. This shows that utilizing the LLMs directly on the SZZ algorithm can not improve the performance too much. For example, LLM4SZZ-raw only improves the F1-score by 4.7% compared to the best baseline R-SZZ and B-SZZ in DS_LINUX and it performs much worse than NeuralSZZ in DS_APACHE. The LLM4SZZ-r variant, which adopts rank-based identification in all scenarios, outperforms LLM4SZZ-raw in F1-score across all three datasets, with improvements ranging from 6.5% to 20.6%. In contrast, LLM4SZZ-re performs worse than LLM4SZZ-r in almost all metrics, indicating that if the LLM cannot comprehend the bug, providing additional context undermines its performance. The table also shows that LLM4SZZ-c, which employs context-enhanced identification in all test cases, improves precision compared to LLM4SZZ-r. However, it may produce empty results in some test cases, leading to lower recall. Additionally, omitting the hint during context-enhanced identification has a significantly negative impact on recall, as evidenced by the performance of LLM4SZZ-h, which is inferior to LLM4SZZ-c in both recall and F1-score.

Overall, LLM4SZZ outperforms all other variants in F1-score, highlighting that the combination of rank-based identification and context-enhanced identification can enhance performance. This also demonstrates the effectiveness of the context-enhanced assessment which evaluates the LLM's capabilities and determines the appropriate identification approach. We also provide the proportions of the two identification approaches in LLM4SZZ across all three datasets. As shown in Figure 7, context-enhanced identification is used more frequently in all three datasets. In DS_LINUX, 64% of the total test cases utilize context-enhanced identification while in DS_GITHUB and DS_APACHE about 60% of test cases adopt this approach.

RQ-2: *The key designs of LLM4SZZ, including the context-enhanced assessment, the context-enhanced identification and the rank-based identification, all contribute to the overall performance. Compared to the rank-based identification, the context-enhanced identification makes a greater contribution. Furthermore, the hint notably enhances the LLM's ability to determine whether a commit contains the bug. Additionally, utilizing LLMs directly in the SZZ algorithm does not significantly improve performance.*

5.3 RQ3. Effectiveness of LLM4SZZ on other LLMs

In this research question, we aim to examine whether the core ideas of our approach (e.g., preparation, context-enhanced ability check, and commits identification) can be applied to other open-source large language models. For evaluation, we choose two additional open-source LLMs: llama3-8b and llama3-70b. The configurations of these two LLMs are described in Section 4.2.

Table 5 presents the performance of different LLMs across three datasets. Among all LLMs, llama3-8b performs the worst. Despite this, it still outperforms all baselines that are not based on LLMs in RQ1, suggesting that our method can be effectively applied to other large language models. Llama3-70b outperforms llama3-8b in all datasets, which is understandable given that llama3-70b

Table 5. The performance comparison between different language models

Model	DS_LINUX			DS_GITHUB			DS_APACHE		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
llama3-8b	0.607	0.536	0.569	0.648	0.595	0.620	0.567	0.371	0.449
llama3-70b	0.645	0.551	0.594	0.657	0.611	0.633	0.612	0.397	0.482
gpt-4o-mini	0.628	0.552	0.588	0.671	0.626	0.648	0.610	0.398	0.482

Fixing Commit: 0b136454741 in linux

After returning from unregister_netdevice_notifier_dev_net(), set the notifier_call field to NULL so successive call to mlx5_lag_add() will function as expected.

1 files changed, 3 addition(+) and 1 deletion(-)

drivers/net/ethernet/mellanox/mlx5/core/lag.c

```

1   if (i == MLX5_MAX_PORTS) {
2-     if (!ldev->nb.notifier_call)
3+     if (!ldev->nb.notifier_call) {
4       unregister_netdevice_notifier_net(&init_net, &ldev->nb);
5+     ldev->nb.notifier_call = NULL;
6+   }

```

Fig. 8. An example where LLM4SZZ fails to choose the correct bug-inducing commit among candidates. It contains more parameters. This also suggests that better LLMs can enhance the performance of LLM4SZZ.

Compared to gpt4o-mini, llama3-70b performs better in DS_LINUX, worse in DS_GITHUB, and similarly in DS_APACHE. In DS_LINUX, llama3-70b outperforms gpt-4o-mini in precision and F1-score. To understand the performance differences, we investigate the test cases where the two models produced different results. We find that llama3-70b tends to be conservative when locating buggy statements and produces an empty result if it is uncertain. For instance, llama3-70b produces results for 1,334 test cases in DS_LINUX, while gpt-4o-mini produces results for 1,373 test cases. In DS_LINUX, gpt-4o-mini and llama3-70b identified almost the same number of true bug-inducing commits, resulting in higher precision for llama3-70b. In DS_GITHUB, many bug-inducing commits can only be found by tracing back unmodified code statements, while llama3-70b tends to conservatively identify deleted lines as buggy statements. Therefore, in DS_GITHUB, gpt-4o-mini outperforms llama3-70b in both precision and recall.

RQ-3: *The core ideas of LLM4SZZ can be applied to other large language models and better LLMs can enhance the performance of LLM4SZZ.*

6 DISCUSSION

6.1 Failure Analysis

In this section, we manually analyze the test cases where LLM4SZZ fails to identify their bug-inducing commits correctly. We randomly select 50 test cases from all failed test cases. After the analysis, we summarize the failed reasons as follows:

Bug-inducing commits cannot be found even by tracing back all lines in the expanded context. As we mentioned above, LLM4SZZ provides LLMs with more context to help them locate buggy statements. If changes are within functions, we expand the context by providing LLMs with the full content of the function. Otherwise, we provide it with the three nearest code statements to the changed lines. However, there are still some bug-inducing commits that cannot be found even by tracing back all lines in the expanded context.

In 27 test cases, LLM4SZZ fails to find bug-inducing commits due to this issue. In eight of these cases, the bug-fixing commits and their corresponding bug-inducing commits modify completely different files. In the remaining cases, the bug-fixing and bug-inducing commits modify the same

Table 6. The performance comparisions between all methods in extended datasets

Method	DS_LINUX			DS_GITHUB		
	Precision	Recall	F1-score	Precision	Recall	F1-score
B-SZZ	0.443	0.592	0.507	0.416	0.669	0.513
AG-SZZ	0.480	0.532	0.505	0.480	0.581	0.526
MA-SZZ	0.407	0.532	0.461	0.401	0.595	0.479
R-SZZ	0.601	0.458	0.519	0.621	0.520	0.566
L-SZZ	0.564	0.430	0.488	0.548	0.459	0.500
LLM4SZZ	0.642	0.579	0.609	0.646	0.604	0.624

files but different functions. Although expanding the context further might help find these bug-inducing commits, an excessively long context will undermine the overall performance of LLM4SZZ, as mentioned earlier.

Failing to locate buggy statements correctly. LLM4SZZ fails to find correct bug-inducing commits in 11 cases due to this issue. One main reason for this is that LLM4SZZ focuses only on modified lines in the context. Sometimes, it is the unmodified code statements in the context that lead to the bug. Although we provide more context to LLMs to avoid this issue, they still face challenges in accurately locating buggy statements.

Failing to choose the correct bug-inducing commit among all candidates. The remaining 12 test cases fail because LLM4SZZ does not identify the correct bug-inducing commit among all candidates. The main reason is that LLM4SZZ fails to determine whether a commit contains the bug. The bug-fixing commit `0b136454741` in Figure 8 is a typical example. From the commit message, LLM4SZZ first locates two buggy statements, lines 2 and 4, resulting in two candidate commits: `e387f7d5fcc` and `7907f23adc1`. Commit `e387f7d5fcc` introduces line 4 and replaces the former function `unregister_netdevice_notifier_dev_net` with the current function `unregister_netdevice_notifier_net`. The LLM determines that although returning from the function `unregister_netdevice_notifier_dev_net` requires setting the `notifier_cal` field to null, this does not imply the same for the function `unregister_netdevice_notifier_net` as they are different functions. Therefore, it determines that commit `e387f7d5fcc^1` does not contain the bug, and LLM4SZZ incorrectly identifies `e387f7d5fcc` as the bug-inducing commit.

6.2 Data Leakage

Since all three datasets were collected before the release of LLMs like GPT-4o-mini [43], there is a potential issue that the performance improvement of LLM4SZZ may result from data leakage. To address this concern, we extend two of the datasets. To extend DS_GITHUB, we follow the same approach as the original paper [47]. Specifically, we iterate through all commits in the dataset's projects, identifying those whose commit messages contain keywords such as "fix" "bug" and "introduce". These commits are added to a candidate list. As GPT-4o-mini's knowledge is limited to data available up to October 2023 [43], we exclude all commits dated before this cutoff. We initially identify 186 candidate commits. Each commit is then manually reviewed to confirm its relevance to bug fixing and to ensure the existence of corresponding bug-inducing commits. After this filtering process, we obtain 148 verified bug-fixing commits and their associated bug-inducing commits. To extend the DS_LINUX dataset, we adopt the method used in the original study [36]. Bug-fixing commits in Linux typically include the keyword "Fixes:" followed by the commit ID of the bug-inducing commit. Using a regular expression, we identify commits with this pattern. Commits dated before October 2023 are excluded, resulting in 9,913 bug-fixing commits. From these, we randomly sample 500 commits for the experiment, ensuring a 95% confidence level and a margin of error under 5%, the same as section 4.2. For DS_APACHE, extending the dataset is

Table 7. Statistics related to the scalability of LLM4SZZ

dataset	llm calls	token numbers	time
DS_LINUX	9.8	14,489	30.43s
DS_GITHUB	10.3	14,890	20.20s
DS_APACHE	13.67	24,023	28.14s

more challenging as it is based on bug reports that lack a fixed format for identifying bug-inducing commits. In the original study, all bug reports were manually analyzed, which required significant human effort. Given that our objective is to demonstrate that the performance improvement of LLM4SZZ is not due to data leakage, the extended datasets of DS_GITHUB and DS_LINUX are sufficient for this purpose. Therefore, we opt not to extend DS_APACHE.

Table 6 presents the experimental results of LLM4SZZ and the baselines for identifying bug-inducing commits across the extended datasets. The best results are highlighted in bold. Among the baselines, the R-SZZ algorithm achieves the best performance in two datasets. In DS_LINUX, R-SZZ performs comparably to the B-SZZ and AG-SZZ algorithms, while in DS_GITHUB, it significantly outperforms all other baselines. From the table, we observe that LLM4SZZ consistently outperforms all baselines. Specifically, in DS_LINUX, it improves precision by 6.8% and the F1-score by 17.3% compared to the best-performing baseline. Similarly, in DS_GITHUB, LLM4SZZ improves precision by 4.0% and the F1-score by 10.2%. These results demonstrate that LLM4SZZ's performance improvements are not caused by data leakage.

6.3 Scalability

Table 7 presents statistics on the scalability of LLM4SZZ, including the average number of LLM calls, the average number of tokens consumed, and the average time required to process a bug-fixing commit. The average number of LLM calls and the average token consumption are primarily determined by the size of the bug-fixing commit. Bug-fixing commits in DS_APACHE are generally larger than those in DS_GITHUB and DS_LINUX, which results in higher LLM call frequencies and greater token consumption for DS_APACHE.

Additionally, the table shows that LLM4SZZ requires approximately 30 seconds to handle a bug-fixing commit. This is longer than the processing time of some baselines, such as the B-SZZ algorithm, because these baselines rely on basic assumptions or simple heuristic rules. However, when compared to more complex techniques like RA-SZZ, which employs program analysis to detect refactorings, LLM4SZZ is significantly faster. Our experiment indicates that RA-SZZ takes an average of 78.4 seconds to process a single bug-fixing commit, nearly 2.6 times longer than LLM4SZZ.

The total time cost of LLM4SZZ consists of two components: the time for LLM calls and the time required to retrieve relevant project information. For example, LLM4SZZ retrieves file contents from specific commits in the project, which contributes to the time cost. This explains why the average processing time for bug-fixing commits in DS_LINUX is longer than that in DS_GITHUB and DS_APACHE, even though DS_LINUX requires fewer LLM calls. The Linux project's large size increases the time required for information retrieval, leading to higher overall processing times. Although this larger size results in increased time, the overall time remains acceptable, indicating that LLM4SZZ is feasible for large projects. We also investigate whether there are any extremely long bug-fixing commits that exceed the LLM's token limit. Our results show that there is only one such commit in the Hadoop project, and it has minimal impact on overall performance.

6.4 Bugs that LLMs fail to understand

In this section, we analyze the test cases in which LLMs fail to understand the bug and fall back to rank-based identification. We randomly select 50 such test cases from the total of 797 failed

Table 8. The statistics of extra bug-inducing commits identified by LLM4SZZ

Count	DS_LINUX	DS_GITHUB	DS_APACHE
only-additions	97	15	3
with-deletions	25	12	6
total	122	27	9

cases and manually examine the characteristics of the bugs. Among these, 38 test cases are from DS_LINUX, 7 from DS_GITHUB, and 5 from DS_APACHE. We categorize the bugs into two types: those with an excessively large context and those requiring subtle changes to resolve.

Bugs involving an excessively large context. LLM4SZZ performs context refinement before requiring the LLM to determine whether the commit contains a bug. However, even after refinement, the context can remain excessively large. In 32 of the analyzed test cases, the context ranges from 330 to 1,887 lines of code. This overwhelming long context hinders the LLM’s ability to accurately detect the bug.

Fixing the bug requires subtle changes. The remaining 18 failed test cases arise from the LLM’s difficulty in detecting subtle changes. For example, some bug-fixing commits only reorder code statements. In such cases, the LLM may incorrectly assume that both versions contain the same statements, misidentifying them as buggy. Similarly, bug fixes involving minimal changes, such as altering a single word in a long string, are also challenging for the LLM to detect.

6.5 Can LLM4SZZ find extra bug-inducing commits?

In this section, we examine whether LLM4SZZ can identify extra commits that all baselines cannot. Table 8 presents the statistics of the extra bug-inducing commits that baselines fail to find. LLM4SZZ identifies 109 extra bug-inducing commits in DS_LINUX, 31 in DS_GITHUB, and 10 in DS_APACHE, accounting for 7.8%, 7.4%, and 2.5% of the total bug-inducing commits, respectively.

We classify these bug-fixing commits into two categories: those containing only added lines and those containing deleted lines. From the table, we observe that LLM4SZZ effectively identifies bug-inducing commits from those with only added lines. Additionally, techniques employed in LLM4SZZ, such as context expanding, facilitate the discovery of extra bug-inducing commits from bug-fixing commits with deleted lines. In DS_LINUX, the majority of extra bug-inducing commits are identified from bug-fixing commits with only added lines. Conversely, in DS_GITHUB, the number of extra bug-inducing commits found from bug-fixing commits with only added lines is nearly equal to those identified from bug-fixing commits with deleted lines.

6.6 Threats to Validity

Internal Validity. The LLMs may produce random outputs during the experiment. To minimize bias, we set the same parameters for all models. Additionally, we repeat the experiment three times and select the majority result as the final output. We also conduct our experiment on large-scale datasets to counteract the randomness. These datasets consist of two programming languages and a total of 2,104 test cases.

External Validity. One potential limitation is that we implement and evaluate LLM4SZZ on only two programming languages, C and Java. However, the majority of bug-fixing commits in the three datasets are written in these two languages. Another concern is that DS_LINUX is created by randomly selecting test cases from the original dataset. To mitigate bias, we selected a total of 1,500 test cases, achieving over a 95% confidence level with a confidence interval of 3. The third threat is the assumption that most bugs are fully fixed and introduced with a single commit. While this assumption holds for most test cases in the three datasets, it may not reflect all real-world scenarios. In the future, we plan to collect additional datasets to address this limitation.

7 RELATED WORK

LLMs in SE. LLMs have been applied to numerous tasks in software engineering [15, 21], such as code generation, software testing and software maintenance. In code generation, researchers have proposed generation models like CodeX [10], AlphaCode [34] and Codegen [42]. They have also improved the performance of code generation using techniques such as chain of thought reasoning [24, 70], static analysis [1] and finetuning [48]. LLMs can also be used to generate new test cases, showing higher coverage [14, 22]. Combing with techniques such as differential testing, they can generate more failure-inducing test cases [33]. In software maintenance, LLMs can be used in tasks such as fault localization [27, 60], bug reproducing [17], bug severity predicting [37] and program repair [62–64].

SZZ algorithm evaluation. The SZZ algorithm and its variants have been extensively evaluated by many researchers. Initially, evaluations are based on datasets manually annotated by researchers [13]. However, building such datasets is time-consuming and may not yield accurate results. To address these challenges, researchers have proposed datasets based on developers' annotations. They extract these annotations from bug reports [59] and commit messages [36, 47].

SZZ algorithm application. The SZZ algorithms have been widely used in empirical studies, including software quality [8], code smells [44], code reviews [4, 29], and developer collaboration [5]. The SZZ algorithm has also been applied to just-in-time defect detection [25, 26, 38]. Researchers use the SZZ algorithm to identify bug-inducing commits in projects, which are then used to train models and evaluate their effectiveness.

8 CONCLUSION AND FUTURE WORK

In this study, we propose a novel approach named LLM4SZZ, which utilizes large language models (LLMs) to locate bug-inducing commits based on bug-fixing commits. The core idea of LLM4SZZ is to adopt different approaches for identifying bug-inducing commits based on the LLM's ability to comprehend the bug. During the ability assessment, we provide the LLM with both expanded and refined contexts to assist it in locating buggy statements and determining whether the bug exists. Based on the assessment results, we then employ either rank-based identification or context-enhanced identification. We evaluate LLM4SZZ using three high-quality datasets, and experimental results show that it outperforms all other baselines in F1-score and can identify extra bug-inducing commits that the baselines cannot detect. In the future, we plan to extend LLM4SZZ to support additional programming languages and collect more high-quality datasets of bug-fixing commits along with their corresponding bug-inducing commits. Additionally, we intend to fine-tune the LLMs to enhance their ability to comprehend bugs and determine their presence in a commit. Furthermore, we aim to leverage the LLMs to identify bug-inducing commits based on bug-fixing commits, even in cases where they do not modify the same files or functions.

9 ACKNOWLEDGEMENT

This research/project is supported by the National Science Foundation of China (No.62372398 and No.72342025) and the Zhejiang Pioneer (Jianbing) Project (2025C01198(SD2)), and funded by ZJU-China Unicom Digital Security Joint Laboratory.

DATA AVAILABILITY

The replication package, which includes the source code, datasets, and LLMs' output, can be found at <https://doi.org/10.6084/m9.figshare.27418236.v1>.

REFERENCES

- [1] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [2] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2019. Empirical Study of Fault Introduction Focusing on the Similarity among Local Variable Names.. In *QuASoQ@ APSEC*. 3–11.
- [3] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*. 2352–2364.
- [4] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 81–90.
- [5] Mario Luca Bernardi, Gerardo Canfora, Giuseppe A Di Lucca, Massimiliano Di Penta, and Damiano Distante. 2018. The relation between developers' communication and fix-inducing changes: An empirical study. *Journal of Systems and Software* 140 (2018), 111–125.
- [6] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).
- [7] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23–581 (2010), 81.
- [8] Bora Çaglayan and Ayşe Başar Bener. 2016. Effect of developer collaboration activity on software quality in two large scale projects. *Journal of Systems and Software* 118 (2016), 288–296.
- [9] Boyuan Chen and Zhen Ming Jiang. 2019. Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering* 24 (2019), 2285–2322.
- [10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [11] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.
- [12] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.
- [13] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139.
- [14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.
- [15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.
- [16] Yuanrui Fan, Xin Xia, Daniel Alencar Da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE transactions on software engineering* 47, 8 (2019), 1559–1586.
- [17] Sidong Feng and Chunyang Chen. 2024. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
- [18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [19] GitPython. 2024. GitPython. <https://github.com/gitpython-developers/GitPython>
- [20] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 200–210.
- [21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).
- [22] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023).
- [23] HuggingFace. 2024. Hugging Face. <https://huggingface.co/>
- [24] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907* (2023).

- [25] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 279–289.
- [26] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.
- [27] Sungmin Kang, Gabin An, and Shin Yoo. 2023. A preliminary evaluation of llm-based fault localization. *arXiv preprint arXiv:2308.05487* (2023).
- [28] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 81–90.
- [29] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yixin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 111–120.
- [30] Cong Li, Zhaogui Xu, Peng Di, Dongxia Wang, Zheng Li, and Qian Zheng. 2024. Understanding Code Changes Practically with Small-Scale Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 216–228.
- [31] Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. 2024. Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 745–766.
- [32] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2023. LooGLE: Can Long-Context Language Models Understand Long Contexts? *arXiv preprint arXiv:2311.04939* (2023).
- [33] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 14–26.
- [34] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweis, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
- [35] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.
- [36] Yunbo Lyu, Hong Jin Kang, Ratnadiira Widayarsi, Julia Lawall, and David Lo. 2024. Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel. *IEEE Transactions on Software Engineering* (2024).
- [37] Ehsan Mashhadi, Hossein Ahmadvand, and Hadi Hemmati. 2023. Method-level bug severity prediction using source code metrics and LLMs. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 635–646.
- [38] Shane McIntosh and Yasutaka Kamei. 2018. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In *Proceedings of the 40th international conference on software engineering*. 560–560.
- [39] Mockus and Votta. 2000. Identifying reasons for software changes using historic databases. In *Proceedings 2000 international conference on software maintenance*. IEEE, 120–130.
- [40] Edmilson Campos Neto, Daniel Alencar Da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 380–390.
- [41] Edmilson Campos Neto, Daniel Alencar Da Costa, and Uirá Kulesza. 2019. Revisiting and improving szz implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–12.
- [42] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
- [43] OpenAI. 2024. gpt-4o-mini. <https://platform.openai.com/docs/models#gpt-4o-mini> Accessed: 2025-02-15.
- [44] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*. 482–482.
- [45] Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl Barr, and Sergey Mechtaev. 2024. The Fact Selection Problem in LLM-Based Program Repair. *arXiv preprint arXiv:2404.05520* (2024).
- [46] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M Germán, and Jesus M Gonzalez-Barahona. 2020. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering* 25 (2020), 1294–1340.
- [47] Giovanni Rosa, Luca Pasarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating szz implementations through a developer-informed oracle. In *2021 IEEE/ACM 43rd International*

- Conference on Software Engineering (ICSE)*. IEEE, 436–447.
- [48] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508* (2023).
 - [49] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.
 - [50] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.
 - [51] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is ChatGPT good at search? investigating large language models as re-ranking agents. *arXiv preprint arXiv:2304.09542* (2023).
 - [52] Lingxiao Tang, Lingfeng Bao, Xin Xia, and Zhongdong Huang. 2023. Neural SZZ algorithm. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1024–1035.
 - [53] treesitter. 2024. tree-sitter. <https://github.com/tree-sitter/tree-sitter>
 - [54] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*, 483–494.
 - [55] Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2017. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process* 29, 1 (2017), e1797.
 - [56] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous graph attention network. In *The world wide web conference*. 2022–2032.
 - [57] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).
 - [58] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
 - [59] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 326–337.
 - [60] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276* (2023).
 - [61] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the plastic surgery hypothesis via large language models. *arXiv preprint arXiv:2303.10494* (2023).
 - [62] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.
 - [63] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).
 - [64] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).
 - [65] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
 - [66] Pengyu Xue, Linhao Wu, Zhongxing Yu, Zhi Jin, Zhen Yang, Xinyi Li, Zhenyu Yang, and Yue Tan. 2024. Automated Commit Message Generation with Large Language Models: An Empirical Study and Beyond. *arXiv preprint arXiv:2404.14824* (2024).
 - [67] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software* 113 (2016), 296–308.
 - [68] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* 48, 1 (2020), 82–101.
 - [69] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608.

- [70] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087* (2023).

Received 31 October 2024; revised 27 February 2025; accepted 31 Mar 2025