Characterizing Regression Bug-Inducing Changes and Improving LLM-Based Regression Bug Detection

Xuezhi Song[†]
Fudan University
Shanghai, China
songxuezhi@fudan.edu.cn

Yijian Wu[*†]
Fudan University
Shanghai, China
wuyijian@fudan.edu.cn

Bihuan Chen[†]
Fudan University
Shanghai, China
bhchen@fudan.edu.cn

Zhengjie Lu[†]
Fudan University
Shanghai, China
zjlu23@m.fudan.edu.cn

Shuning Liu[†]
Fudan University
Shanghai, China
liushuning@fudan.edu.cn

Xin Peng[†]
Fudan University
Shanghai, China
pengxin@fudan.edu.cn

Abstract

Regression bugs are common in real-world software projects. Although several studies have characterized various aspects of these bugs, a detailed analysis of the characteristics of code changes that introduce regression bugs is still lacking. It is also not clear whether and how regression bugs can be identified, in the era of wide usage of large language models (LLMs), in code commits during software development in the first place. To fill this gap, our study systematically analyzes 280 regression bugs from open-source Java projects, examining both the root causes and the associated development activities of regression bug-inducing changes, while also evaluating the effectiveness of LLMs in detection such commits and explaining their underlying causes.

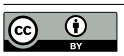
Our findings indicate that regression bugs are usually triggered by a single atomic development activity, with feature changes or additions, and bug fixes appearing more frequently in regression bug-inducing commits. Additionally, performance improvements and refactoring are often responsible for the introduction of bugs. We develop a taxonomy that categorizes the root causes of regression bugs into two types: intrinsic errors, such as logic errors and unchecked null references, and compatibility errors, where otherwise correct changes inadvertently violate assumptions or dependencies in other parts of the system.

Furthermore, we verify that LLMs have limited ability to detect regression bugs. Based on our findings, we construct LLM4Reg that substantially improves the precision, recall, and explanation capabilities of LLM-based regression bug detection.

CCS Concepts

• **Software and its engineering → Software evolution**; **Maintaining software**; **Software testing and debugging**.

[*]Corresponding author
[†]Also affiliated with Shanghai Key Laboratory of Data Science.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773182

Keywords

Delta Debugging, Critical Changes, Probabilistic Model

ACM Reference Format:
Xuezhi Song, Yijian Wu, Bihuan Chen, Zhengjie Lu, Shuning Liu, and Xin Peng. 2026. Characterizing Regression Bug-Inducing Changes and Improving LLM-Based Regression Bug Detection. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773182

1 Introduction

Regression bugs represent a unique category of software defects that break existing functionalities. As a typical side effect, regression bugs are usually introduced when developers make various types of source code changes such as adding new features or code refactoring [54]. Due to incompleteness of tests and continuous changes in requirements, developers may not be aware of the broken functionalities at the time of code changes until they encounter unexpected behaviors or failures on some previously working functionalities.

Fixing regression bugs is difficult and labor intensive because it is unclear when the bug was introduced and there could be a large amount of code changes after the (unawared) introduction of the bug [4]. Developers have to put a lot of efforts into identifying when and why regression bugs occur and how to fix them by examining the code changes in the history. Moreover, regression bugs cause a negative impact on the end-user experience, which may be more annoying than non-regression bugs since the previously working functionality is broken.

Regression bugs frequently occur in real-world software projects. Research has reported that typically almost half of all bugs in a software project are regression bugs [24, 57]. Several empirical studies [1, 5, 15, 24, 56, 61] have been conducted to understand regression bugs from various aspects, such as change complexity, category, lifespan, and priority, as well as how development activities introduce bugs. The results of these studies offer valuable insights that drive progress in related research areas, such as regression testing [3, 13, 27, 32], fault localization [7, 18, 19, 25, 33, 48, 49, 60, 62, 63], and automated program repair [14, 22, 44].

However, these studies still lack a comprehensive analysis of the characteristics of code changes that introduce regression bugs

in real-world software commits, particularly with respect to the types of root cause in code changes. For example, the widely used regression categories from [15] consider only whether the impact of the changes is confined to a single module. There were also studies [1, 61] that investigate how refactoring and bug fixes might trigger regression bugs, but they did not systematically explore other development activities. A comprehensive understanding of the characteristics of code changes that introduce regression bugs is essential for automatic detection of regression bugs and for facilitating subsequent fixes.

LLMs are widely used for defect detection in software development processes [28, 31, 45, 50]. Researchers have constructed various empirical studies and approaches to improve the performance of LLMs in defect detection. However, to the best of our knowledge, there is limited research exploring the capability of LLMs in detecting regression bugs in code commits. Given the remarkable capabilities that LLMs have demonstrated in code understanding and analysis, exploring their potential in regression bug detection is of great significance.

To address these gaps, we conducted the comprehensive study to characterize regression bug-inducing changes in open-source Java projects and evaluated the effectiveness of LLMs in detecting these bugs. To this end, we performed a detailed manual analysis on 280 regression bugs collected from 84 high-rated open-source Java projects. To the best of our knowledge, it is the largest regression bug dataset in which each bug has been thoroughly analyzed for the purpose of understanding the rationale of bug-inducing and various characteristics.

To comprehensively understand regression bug-inducing commits(BICs), we designed three research questions (RQs).

- **RQ1:** What development activities are observed in the BICs of regression bugs? Are there any development activities that are more likely to cause regression bugs in all BICs?
- **RQ2:** What are the characteristics of regression bug-inducing changes? Specifically, what are the root causes of regression bugs, what is the relationship between bug-inducing changes and bug-fixing changes?
- **RQ3:** Can current LLMs effectively identify regression bugs?

Our empirical study reveals that regression-inducing commits (BICs) often contain multiple development activities, yet typically only one activity—most frequently feature additions or changes, and bug fixes—actually introduces the regression bug. Among these activities, performance enhancements and bug repairs are most likely to be responsible for regressions. We further observe that approximately two-thirds of regression bugs arise from intrinsic errors, while the remaining one-third are due to compatibility issues. Moreover, the code segments requiring adaptation tend to lie at a considerable distance from the modified regions. When evaluated out of the box, large language models (LLMs) demonstrate limited capability in both identifying and explaining regression bugs.

Based on our empirical findings, we present LLM4Reg, a novel tool that leverages commit decomposition to mitigate the performance degradation caused by excessive input tokens, then implements root-cause-specific workflows that initially detect intrinsic errors under the assumption of correct context, followed by compatibility error detection based on information from change impact analysis and co-change pattern analysis when no intrinsic errors are detected. Experimental evaluation demonstrates that LLM4Reg significantly outperforms the state-of-the-art LLM baseline, with F1-score improved from 0.3940 to 0.5094 (+29.3%), and explanation success rate improved by 27.5% (14.67 *pts*).

In summary, our work makes the following main contributions.

- We constructed a benchmark datasetcomprising 280 human-curated regression bugs, each of which is annotated with minimized bug-inducing activities, line-level bug-inducing changes, and detailed root cause and rationale that triggers the bug.
- We conducted an empirical study to characterize regression inducing changes, highlighting the importance of decomposing and categorizing development activities. Furthermore, we categorized the root causes of regression bugs and identified various types of knowledge required to understand the bugs.
- We found the weakness of SOTA LLMs in identifying and explaining regression bugs and conducted experiments to demonstrate that our findings can effectively improve the performance of LLMs in identifying and explaining regression bugs.

2 Methodology

We first describe the details of our data collection and annotation, and then elaborate the setup of each of the three research questions introduced in Section 1.

2.1 Data Collection

Our empirical study is based on a large dataset of regression bugs, which contains 1,025 regression bugs. Each regression bug includes a bug-inducing commit (BIC) and a bug-fixing commit (BFC), accompanied with the test code that comes with the bug fix to validate the fix. These regression bugs were automatically collected from open source projects by RegMiner [41], which ensures that the test code can be migrated to target revisions and satisfies all the following criteria: (1) tests pass at vBFC, (2) tests fail at vBFC-1, (3) tests fail at vBIC, and (4) tests pass at vBIC-1, where vBFC and vBIC stand for snapshot versions *after* BFC and BIC, respectively, and -1 stands for the snapshot version *before* the specific commit. Details of the data set construction process can be found in the literature [41]. We further calibrate the critical changes of bug-inducing and bug-fixing with the help of C2D2 [42] to improve the quality of data.

We realized that comprehensively understanding 1,025 regression bugs by manual inspection is infeasible for our annotators. Therefore, we randomly sampled 280 regression bugs. The sample size was determined according to standard statistical formulas [43] to achieve a 95% confidence level at a ±5% margin of error, so that the proportions estimated from the sample can reliably reflect those of the entire dataset. Since this dataset contains only regression bug samples, we named it the Regression Sample Dataset (RSD).

As our goal is to analyze why regression bugs are introduced, we formulate a positive sample dataset (PSD) by selecting only the BICs (including vBIC-1, vBIC, and the bug-inducing changes) from the RSD. As commits that do not introduce bugs are much more numerous than bug-inducing commits in real-world development (typically 10:1 according to the literature [35]), we supplemented

our dataset with 2,800 manually labeled non-BICs randomly selected from JIT-Defects4J [35] to fairly evaluate LLM performance. Combined with the positive samples from RSD, the Full Sampled Dataset (FSD) contains 3,080 commits in total.

By this means, we formulate two original datasets. The RSD containing information of 280 regression bugs is used for data annotation and analysis to answer RQ1 and RQ2. The FSD containing information of 280 BICs (taken from RSD) and 2,800 non-BICs is used to answer RQ3.

2.2 Data Annotation

In data annotation, the key role is the annotator who strictly follows a protocol to process each regression bug and the corresponding source code changes in BIC and BFC. The protocol pipeline is defined as the following five stages to deal with each regression bug in the RSD.

- *Stage 1*: Decomposing each BIC into multiple development activities.
- *Stage 2*: Identifying the critical bug-inducing changes in BIC.
- *Stage 3*: Identifying the critical bug-fixing changes in BFC and marking the spatial relationship to the critical bug-inducing changes.
- *Stage 4*: Labeling the type of root cause.
- *Stage 5*: Determining the development activities responsible for inducing bugs, based on the critical bug-inducing changes.

Two graduate students were recruited as annotators. Both of them were majored in software engineering and had more than three years of Java development experience, satisfying the requirement of foundational programming and debugging skills for this task. They were asked to strictly follow the annotation protocol and independently annotate the given regression bugs. To evaluate agreement throughout the annotation process, we continuously monitored agreement level between the two annotators using Cohen's Kappa coefficient [47]. Reconciliation had been conducted after each 10% increment, i.e., at the completion of 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100% of the total BIC annotations. For cases with notable disagreements, a senior supervisor with rich software development experiences would organize a comprehensive discussion with the two annotators on the rationale behind the regressions. By this means, we resolved the disagreement between the annotators and ensured that every regression bug received credential annotations. Finally, the Cohen's Kappa values for inter-rater reliability across stages 1 to 4 were 0.8542, 0.8585, 0.9338, and 0.8396, respectively.

2.3 Setup of Development Activities Analysis (RQ1)

We define *development activities* based the atomicity of commits according to the industrial recommendation defined in Conventional Commits[1]. A set of source code changes serving for a single purpose are regarded as an atomic commit. In real world development, however, a commit may not strictly follow such a recommendation. Multiple purposes, such as introducing new feature, refactoring, or fixing bugs, may be mixed in a single commit. To analyze development activities within BICs, we first define a list of types of atomic commits in terms of general purposes of the source code changes, as shown in Table 1. This list is similar to the commit types as recommended in the Conventional Commits, with slight alternatives to adapt to our empirical study.

Then the annotators were asked to comprehend the purposes of source code changes in each BIC and break down the set of changes into groups so that each of the groups could form up an atomic commit. Since it is challenging to understand and reorganize possibly numerous changes in one commit, the annotators are allowed to use SmartCommit [40], an interactive assisting tool that automatically divides the changes into smaller single-purposed commits. Although it is not designed for development activity analysis, it is helpful for the annotators to quickly understand the intentions of the changes and further provide benefits in the decomposition of the commits and the annotation of development activity type for each decomposed atomic commits.

Finally, all changes in source code in the BICs in RSD were decomposed into groups, each of which serves a single purpose defined in Table 1. Furthermore, the annotators would later take a note of which development activity actually triggered the regression bug after identifying the root causes of the regression bug in the next analysis step (see Section 2.4).

Table 1: Development Activity Types

Label	Description
feature	Introduces new features or enhances existing ones.
fix	Fixes bugs.
docs	Modifies documentation only.
style	Changes code formatting without affecting code execution.
refactor	Refactors code without adding new features or fixing bugs.
test	Adds or modifies tests.
perf	Changes code to improve performance.
build	Changes affecting the build system or external dependencies.
ci	Changes related to continuous integration.
chore	Other changes that do not modify source or test files.
revert	Reverts a previous commit.

2.4 Setup of Regression Bug Characteristics Analysis (RQ2)

We characterize the regression bugs from the following perspectives, i.e., root cause, knowledge required to understand the bugs, and the spatial relationship between bug-inducing and bug-fixing code.

Two annotators conducted independent labeling tasks using an open-coding scheme[2] for all perspectives above. The annotators were asked to freely explore the source code, take notes of anything they found, and document their reasoning and the steps taken to finalize the bug localization. They were also asked to give labels on each perspective. Some initial instinctive labels had been provided. For example, we provide initial root causes of bugs reported by [15], initial knowledge types (domain-specific knowledge and project-specific knowledge) along with their definitions, and initial spatial

[1] https://www.conventionalcommits.org/en/v1.0.0/#specification

[2] https://atlasti.com/research-hub/open-coding

relationships between bug-inducing and bug-fixing changes as used by Wen et al. [53].

During the analysis process, each annotator had access to a web-based tool to review the source code changes and browse any files required to understand the bug. They may also use an integrated development environment (IDE), such as IntelliJ IDEA, in which the software project under consideration could be loaded and the source code could be executed and debugged. They might also turn to Large Language Models (LLMs) such as ChatGPT or intelligent coding assistants like Copilot for help in their annotation work. They were also allowed to use search engines and project documentation to find anything that might be helpful for them to understand the source code related to the regression bugs.

Disagreements were periodically reviewed, discussed, and resolved. For cases with notable disagreements, a senior supervisor with rich software development experiences was involved to resolve the discrepancies through a discussion of the localization process and bug characteristics, ultimately leading to a consensus. The inter-rater reliability, measured in Cohen's Kappa, was 0.8901.

Since this work was extremely mental work intensive, the annotators were asked to be fully engaged as they were doing the analysis and to limit their analysis work within 6 hours per working day. Typically, an annotator had to spend 1-2 hours to comprehend a regression bug. Besides intensive cognitive work, the annotation process required iterating annotation protocols, conducting consistency checks, and multiple rounds of consensus discussions, costing nearly 10 person-months in total.

2.5 Setup of Regression Bug Detection Evaluation (RQ3)

To effectively evaluate LLMs' performance on regression bug detection, we selected several SOTA LLMs, including GPT-4o, Claude Sonnet 3.5, and DeepSeek-V3, which represent the current large-scale parameter models. In addition, we included codellama models (7B, 13B, and 34B parameters) and StarCoder2-15B to further assess the relative performance of smaller specialized code models.

Table 2 presents detailed information about the models, including parameter counts, maximum token limits, and knowledge cutoff dates of the large models used. Here, "Type" indicates whether the model is accessed via a remote API or deployed locally. Local models were deployed on an Ubuntu 20.04 server equipped with a 16-core, 32-thread Silver 4208 CPU (2.10 GHz), 64 GiB of RAM, and four NVIDIA GeForce RTX 3090 GPUs, each with 24GB memory.

Table 2: Ranges of parameter numbers, max. token limits, and training knowledge cut-offs

Model	#Params	Max. Tokens	Type	Knowledge Cut Off
GPT-4o	-	128K	API	2023/10
Claude Sonnet 3.5	-	200K	API	2024/04
deepseek-V3	671B	128K	API	2024/07
codellama-7b-hf	6.74B	100K	Local	2022/01
codellama-13b-hf	13B	100K	Local	2022/01
codellama-34b-hf	33.7B	100K	Local	2022/01
starcoder2-15b	15B	16K	Local	2023/11

For each model, the precision, recall, and F1-score were evaluated for comparison. Each model also provided explanations for the detected regressions. The annotators read the output explanations of the detected true-positive regressions and decided whether the explanations were understandable and correct. The explainability was calculated as the proportion of all understandable and correct explanations out of all true-positives, and recorded for comparison.

3 Development Activities Analysis (RQ1)

3.1 Overall Distribution of Activities

Figure 1 illustrates the number of occurrences of each development activity among the 280 BICs. For each type of development activity, we count the total occurrences (shown as the light-green bar) and the number of occurrences which are directly related to bug inducing (shown as the heavy-green bar). We identify total 1,200 occurrences of individual development activities, which is averagely 4.29 activities per BIC. This means that multiple development activities are usually mixed in a single commit, increasing the complexity of commits and the risk of introducing regression bugs.

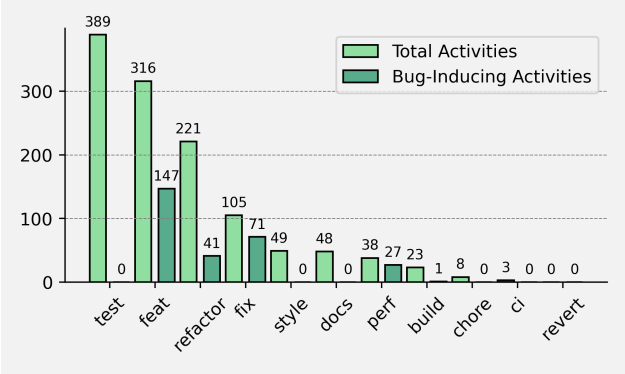


Figure 1: Development Activities in 280 BICs

However, the average number of bug-inducing activities per BIC is only 1.03, indicating that regression bugs are typically caused by a *single* development activity. Only in seven BICs we find that bugs were introduced through a combination of *refactor* and either *feature* or *fix*.

The most frequent development activities in BICs are *test* (389), *feature* (316), and *refactor* (221). Notably, while *test* activities are highly frequent and broadly present in BICs, they rarely introduce bugs since their primary purpose is to verify functionalities. In contrast, *feature* and *fix* emerge as the most significant contributors to regression bugs, with 147 and 71 respectively. *refactor* and *perf* also contribute meaningfully, with 41 and 27 respectively. By comparison, *build* are rarely responsible, with only one case.

Finding 1: Regression BICs typically include multiple development activities. However, it is almost a single specific activity that ultimately causes the regression bug. Changes related to new features and bug fixes are the most frequent activities related to regression bug introduction.

3.2 Bug-Inducing Activity Analysis

Given a regression bug, we further analyze the probability that it was induced by a specific type of development activity. The probability[3] is calculated as the proportion of bug-inducing activities among all activities of the same type in BICs.

Although the *perf* appeared only 27 times among bug-inducing activities, it demonstrates the highest probability at 71.05%. Similarly, the *fix* exhibits a probability of 67.62%. While the *feature* label is the most frequent among bug-inducing activities, it shows a probability of 46.52%, which is lower than those of *perf* and *fix* but remains substantial. In contrast, only 18.55% and 4.35% of *refactor* and *build* activities induce bugs, respectively.

These results suggest that when a regression bug occurs, developers should pay more attention to *perf*, *fix*, and *feature* activities, as they are associated with a significantly higher likelihood of being responsible for regression.

Finding 2: When a regression bug is induced, *perf*, *fix*, and *feature* activities are more likely to be responsible, with probabilities of 71.05%, 67.62%, and 46.52%, respectively.

3.3 Testing Activities Analysis

We find that *test* is the most frequent development activity, appearing 389 times. Our further investigation reveals that 78.21% of BICs contain test addition or modification, with an average of 1.78 tests per commit. This means that each commit typically introduced tests targeting around two different functionalities. Despite this thorough testing, regression bugs were still introduced.

To further understand this phenomenon, two annotators manually check the differences between tests added or modified in BICs, referred to as BIC Tests, and tests added or modified in bug-fixing commits, referred to as Regression Tests. The results of this analysis are as follows.

- *Missing Test Inputs* (45.66%). Compared to Regression Tests, BIC Tests overlook certain specific inputs of the feature or function under test. For example, in the case of univocity-parsers-420[4], the commit added support for CSV data with spaces as delimiters, as well as test inputs using spaces as delimiters. However, it failed to consider data that include both commas and spaces as delimiters, leading to the introduction of bugs.
- *Missing Test Features* (43.38%). BICs sometimes lack tests for some features or combinations of multiple features, or miss certain sequences of tests if there are dependencies between tests. For example, when object serialization is changed, the test on the corresponding changes in deserialization function is missing.
- *Specification Inconsistency* (10.96%). There are discrepancies between the specifications of Regression Tests and BIC Tests, mainly reflected in different assertion conditions. This category suggests a misunderstanding of requirements by the test developers.

On the other hand, in 21.79% of total BICs, no tests were added or modified. In 65.57% of these cases, the project did not include any tests targeting the feature which was implicitly broken by the changes. In 11.48% of the cases, there were tests that protected the target feature, but some test input was missing. In 16.39% of the cases, the existing tests in the project were outdated. The rest 6.56% contained correct tests and the corresponding test inputs for the broken feature but the regression bug were still introduced due to incomplete execution of the tests.

Finding 3: Even though developers added or modified tests in 78.21% of the BICs, regressions still occurred. The main reasons were insufficient diversity in test data types, missing tests for regressed features, and misunderstanding of specifications by developers. Furthermore, in some BICs where tests were neither added nor changed, the existing tests were outdated.

3.4 Implications

Finding 1 and Finding 2 suggest that, for effective identification and explanation of regression bugs, we could consider decomposing commits, such as SmartCommit [40], and paying adequate attention to different development activities. For example, when using LLMs, decomposing commits can mitigate issues associated with excessive input, such as difficulty in focusing on the most relevant changes due to too much detail, as well as the risk of exceeding the model's input limit and losing important context. Note that this strategy is equally applicable to traditional bug-inducing detection tasks [9, 16, 17, 35, 38, 59, 64]. Such tasks typically extract structured features from commits, such as whether the commit is a bug fix, lines of code added/deleted, and other metrics. Through finer-grained decomposition of commits, more precise feature representations can be obtained.

Finding 1 can also optimize classical regression bug localization techniques such as Delta Debugging (DD) [62, 63]. Hierarchical DD [25, 33] approaches use hierarchical strategies for large-scale changes, such as performing DD first at the file level, then at the hunk level. Since Finding 1 shows that the vast majority of regression bugs are induced by sub-commits containing a single atomic activity, future work could target a new hierarchy level: commit decomposition. The enhanced DD process would first decompose commits into atomic activity sub-commits, then directly search these sub-commits to locate the regression bug-inducing one.

Finding 3 indicates that during the software development process, developers should invest more effort in testing potentially affected functionalities, constructing more comprehensive testing scenarios, and even examining existing, possibly outdated tests. However, this implies higher testing costs, and developers can explore alternative approaches, such as leveraging LLMs, to further enhance the detection of potential regression bugs.

[3]This is a conditional probability P(bug-inducing activity|BIC). We did not compute the conditional probability P(BIC|activity), which may answer the question "which activity is more likely to introduce regression bugs in general". Calculating P(BIC|activity) is a non-trivial task that would require to (1) identify and label code changes in all commits from a wide range of projects by development activities and (2) determine whether each of those changes introduced a regression bug. It is practically infeasible due to the prohibitive investment of manual effort and time overhead required. Instead, for the purpose of this study, we intend to show that, within the range of all BICs, how the activities distribute and which activities are more related to regression introduction.
[4]https://github.com/uniVocity/univocity-parsers/issues/420

4 Regression Bug Characteristics Analysis (RQ2)

4.1 Root Causes

As shown in Table 3, we categorized the root causes of regression bugs into the following two types based on the characteristics of bug-inducing changes.

- *Compatibility Errors* (36.43%). Compatibility errors refer to the situations where the changes made in BIC are technically correct but other parts of the source code in the project are incompatible with these changes, which ultimately lead to a regression bug.
- *Intrinsic Errors* (63.57%). Intrinsic errors refer to the situations where the changes in BIC are erroneous by themselves. In other words, the root cause does not fall in other parts of the source code but is limited to the scope of the source code changes in the commit alone.

Table 3: Root Cause and Location With Fixing Changes on 280 BICs

Root Cause	Location With Fixing Changes				
	#Loc	#Meth	#File	#Pack	#Diff Pack
Compatibility Errors (102)	0	1	30	27	44
Data Flow Incompatibility (62)	0	1	16	12	33
Invocation Incompatibility (40)	0	0	14	15	11
Intrinsic Errors (178)	75	91	11	1	0
Business Error (138)	52	78	7	1	0
Unchecked Null (16)	7	7	2	0	0
API Misuse (13)	10	1	2	0	0
Uncaught Exception (8)	5	3	0	0	0
Cast Error (3)	1	2	0	0	0
Total (280)	75	92	41	28	44

Our classification approach differs from a prior work [15], which categorizes regression bugs based on impact scope[5]. Instead, we focus on the nature of the defective code: intrinsic errors occur when the buggy code is within the change itself, while compatibility errors arise when the buggy code exists in other parts of the project that fail to adapt to an otherwise correct change. This distinction has important implications for defect detection and repair strategies. For example, detecting compatibility errors may require examining multiple parts of the project for code incompatible with the changes, and different repair strategies can be tailored based on the error category.

For compatibility errors, we further identified two subcategories.

- *Invocation Incompatibility* (39.22%): This type refers to the cases where the regression bug-inducing changes are incompatible with some other functions or methods that are directly or indirectly invoked by the modified code. In other words, existing functions or methods do not handle the newly introduced behavior of the modified code, and thus break existing functionality.
- *Data Flow Incompatibility* (60.78%): This type refers to the cases where the bugs are not directly related to any caller-callee relationships but only the changes in data flow lead to incompatibility between different components of the source code and cause a bug.

For intrinsic errors, we identified five subcategories.

- *Logic Error* (77.53%): This is the most common subcategory. Logic error occurs when the logic implemented in the code changes is erroneous, leading to incorrect functionality or unexpected behavior. This also includes the cases where certain changes are missing when multiple changes should have been made.
- *Unchecked Null* (8.99%): This type occurs when the code fails to check for null values before dereferencing objects, potentially leading to a "NullPointerException" at run-time.
- *API Misuse* (7.30%): This type occurs when a developer incorrectly uses an API, e.g., calling a method with wrong parameters, or misunderstanding the API's behavior.
- *Uncaught Exception* (4.49%): This type occurs when exceptions are not properly handled, resulting in crashes or unintended program termination.
- *Cast Error* (1.69%): This type occurs in cases where an incompatible type cast occurs, leading to a `ClassCastException`. Only three cases are in this subcategory.

Finding 4: Nearly two-thirds of the regression bugs were caused by intrinsic errors, whereas the rest were caused by compatibility errors. Most (more than 3/4) intrinsic errors were related to sophisticated logic, while less than 1/4 were related to coding defects.

The results highlight the complexity of regression bugs. Business errors and compatibility errors together account for 85.71% of the total. Both types are challenging to detect and localize. First, business errors constitute nearly half of the cases. Analyzing such bugs typically requires clear specifications in project, but these specifications are often diverse. For example, commit 8ca9472[6] introduced a regression bug: it broke the specification for parsing comments, which does not allow formatting of the original comments. However, in other projects where the specification does allow formatting, this bug would not occur.

Second, compatibility errors represent a significant proportion. Although it might seem intuitive that invocation incompatibility and data flow incompatibility could be identified using program analysis techniques like constructing call graphs and performing data flow analysis, complex compatibility scenarios still exist and prove traditional program analysis techniques ineffective. For example, I/O data flow, runtime class loading, and reflective annotation processing create implicit dependencies difficult to trace statically. We used the static code analysis tool Spoon [37] to construct a Program Dependence Graph for the snapshot of the project code corresponding to each bug, and found that only 49.02% of regression bug code changes are reachable from the incompatible code.

JEXL-202[7] is a real regression bug in Apache Commons JEXL[8], caused by data flow incompatibility. Figures 2 and 3 illustrate the bug-inducing and bug-fixing changes, respectively. In the BIC, multiple assignment operators were added to the `JexlOperator` class, such as `selfAdd` (Line 12 in Figure 2) and `selfSubtract` (Line

[5]This work [15] defines two categories: *local* a change introduces a new bug in the changed module or component, and *remote* a change in one part of the software breaks functionality in another module or component

[6]https://github.com/getgauge/gauge-java/commit/8ca94723ca24278fd103ba6e446d171d5a46eb9d
[7]https://issues.apache.org/jira/browse/JEXL-202
[8]https://github.com/apache/commons-jexl

```
public enum JexlOperator {
/**
 * <strong>Syntax:</strong> <code>x + y</code>
 * <br><strong>Method:</strong> <code>T add(L x, R y);</code>
 * @see JexlArithmetic#add
 */
  ADD("+", "add", 2),
/**
 * <strong>Syntax:</strong> <code>x += y</code>
 * <br><strong>Method:</strong> <code>T selfAdd(L x, R y);</code>
 */
+ SELF_ADD("+=", "selfAdd", ADD),
/**
 * <strong>Syntax:</strong> <code>x -= y</code>
 * <br><strong>Method:</strong> <code>T selfSubtract(L x, R y);</code>
 */
+ SELF_SUBTRACT("-=","selfSubtract",SUBTRACT),
/**
 * <strong>Syntax:</strong> <code>x *= y</code>
 * <br><strong>Method:</strong> <code>T selfMultiply(L x, R y);</code>
 */
+ SELF_MULTIPLY("*=","selfMultiply",MULTIPLY),
+ ...
```

Figure 2: Bug-inducing changes for JEXL-202

```
 // The set of assignment operators as classes.
+ private static final Set<Class< ? extends JexlNode>> ASSIGN_NODES
  = new HashSet<Class< ? extends JexlNode>>(
+   Arrays.asList(
+   ASTAssignment.class,
+   ASTSetAddNode.class,
+   ASTSetSubNode.class,
+   ASTSetMultNode.class,
+ ...
@@ -186,7 +205,7 @@ void jjtreeCloseNodeScope(JexlNode node)
throws ParseException {
...
} else if (node instanceof ASTAmbiguous) {
throwParsingException(
JexlException.Ambiguous.class, node);
-  } else if (node instanceof ASTAssignment) {
+  else if  (ASSIGN_NODES.contains(node.getClass())) {
     JexlNode lv = node.jjtGetChild(0);
     if (!lv.isLeftValue()) {
     throwParsingException(
     JexlException.Assignment.class, lv);
```

Figure 3: Bug-fixing changes for JEXL-202

17 in Figure 2), resulting in new assignment Abstract Syntax Tree (AST) nodes like `ASTSetAddNode` and `ASTSetSubNode`. This change broke the code in Figure 3 (Line 17) responsible for determining the assignment node types. In this case, the program dynamically generates assignment-related AST node classes at runtime based on enumeration types from `JexlOperator`, which are then utilized in other parts of the project. This indirect propagation hinders the effectiveness of data flow analysis in accurately tracing the relevant dependencies.

4.2 Knowledge Required to Understand Regression Bugs

To better understand the characteristics of regression bugs, we further investigated the types of knowledge required by participants to locate and comprehend their root causes. The results encompass three types of knowledge required to fix a regression bug: Programming Knowledge, Domain-Specific Knowledge, and Project-Specific Knowledge.

- *Programming Knowledge* consists of general programming concepts and practices that apply across various coding contexts. This includes foundational principles like understanding that uninitialized objects can lead to NullPointerException, or recognizing common coding patterns that prevent such errors.
- *Domain-Specific Knowledge* refers to specialized knowledge unique to a specific field and applicable across projects within that domain. For example, in Java dynamic analysis projects, domain knowledge includes the meanings of various Java bytecode. Additionally, it encompasses cross-project insights, such as recurring issues, common patterns in code changes, and expertise in widely used third-party libraries.
- *Project-Specific Knowledge*, on the other hand, involves in-depth familiarity with the architecture of an individual project. This includes understanding the specific functions and return values of interfaces, established usage patterns of code and APIs, and typical change patterns unique to that particular project.

The results show that among all bugs, only 7.50% could be understood and localized using programming knowledge alone, 22.14% required additional domain-specific knowledge, 32.14% required project-specific knowledge, and 38.21% required a combination of both project- and domain-specific knowledge.

Specifically, for compatibility errors, *zero* bug could be understood and localized with programming knowledge alone, and only 5.88% required additional domain-specific knowledge together with programming knowledge. The majority still required project-specific knowledge or a combination of both project- and domain-specific knowledge.

For intrinsic errors, nearly 60% required project-specific knowledge or a combination of project- and domain-specific knowledge, while 31.46% needed additional domain-specific knowledge beyond programming knowledge. Only 11.80% of the bugs could be understood with programming knowledge alone. Although intrinsic errors do not require the location of incompatible code within the project, they still require project-specific knowledge to provide the necessary context for understanding the bugs.

Finding 5: Most regression bugs (70.35%) require either project-specific knowledge or a combination of project- and domain-specific knowledge for effective understanding and localization, with compatibility errors especially dependent on this knowledge. Although intrinsic errors often require less contextual information, nearly 60% still rely on project-specific knowledge to fully understand the bug context.

4.3 Regression Bug Inducing and Fixing Locations

Understanding the spatial relationship between regression bug-inducing changes and regression bug-fixing changes can significantly enhance bug localization research and provide insight into how much additional context beyond the changes themselves is

necessary for effective bug detection. We denote regression bug-inducing changes as C_{induce} and regression bug-fixing changes as C_{fix}. As shown in Table 2, we categorize their relationships into five types:

- *Same Location* (26.79%): C_{induce} and C_{fix} occur at the same location.
- *Same Method* (32.86%): C_{induce} and C_{fix} at different locations but within the same method.
- *Same File* (14.64%): C_{induce} and C_{fix} are in different methods but within the same file.
- *Same Package* (10.00%): C_{induce} and C_{fix} are in different files but within the same package.
- *Different Packages* (15.71%): C_{induce} and C_{fix} are in different packages.

For compatibility errors, C_{induce} and C_{fix} are located within the same file in 31 cases, within the same package in 27 cases, and across different packages in 44 cases. The fact that 69.61% of these cases are either within the same package or in different packages indicates that C_{induce} and C_{fix} often occur at considerable distances from each other within a project.

For intrinsic errors, C_{induce} and C_{fix} are located at the same location for 75 cases, and within the same method for 91 cases. However, in seven cases, they are located in different files within the same package, and in one case, even in a different package. We found that these bug fixes are typically workarounds.

Finding 6: The spatial distribution between regression bug-inducing and regression bug-fixing changes varies significantly between error types, with compatibility errors often occurring at greater distances. In contrast, intrinsic errors are typically resolved close to inducing changes.

This results show that bug detection for bugs in the compatibility errors category is a challenging task, partly because it is difficult to determine the exact distance between related codes. On the other hand, we may end up with a large amount of contextual information, and handling such extensive context effectively is also a challenge.

Additionally, for intrinsic errors, the changes themselves can often serve directly as locations for fixes. This allows us to effectively reframe the fault localization task as identifying bug-inducing changes. This result also explains why previous work [52] has been able to use bug-inducing changes to improve fault location methods.

4.4 Implications

Findings 4 and 6 suggest that the detection and repair of regression bugs can benefit markedly from distinguishing between intrinsic errors and compatibility errors. This distinction enables differentiated strategies: intrinsic errors can often be detected with only limited context around the code change, whereas compatibility errors typically require a broader analysis of related components. Furthermore, leveraging classification results enables developers to rapidly locate repair positions.

The same distinction is also essential when training language models for regression bug detection. Intrinsic errors can usually be represented directly by the changed code, whereas compatibility errors call for designating incompatible code segments as buggy code. Treating the two error types separately tends to yield more effective models.

Despite these benefits, Findings 4 and 6 also show that the detection and repair of regression bugs remains a challenge. The high proportion of logic and compatibility errors greatly raises overall complexity, and explicit functional specifications are often unavailable. Examining a feature's evolution in the version history may help infer its intended behavior. Compatibility errors are especially problematic. Locating incompatible code typically requires scanning and analyzing a large volume of source files to uncover long-range dependencies. Future work should therefore focus on techniques that can retrieve and integrate such critical contextual information more efficiently.

Furthermore, Finding 5 indicates that regression bug detection benefits from knowledge retrieval, particularly project knowledge and domain knowledge. This means that future work could consider integrating search engines, similar domain code retrieval, and project code retrieval functionality. Project code retrieval overlaps to some extent with the suggestions from Finding 6.

Table 4: Comparison of Performance Metrics for Different LLMs on FSD

Model	Precision	Recall	F1-score	Explain.
GPT-4o	0.1425	0.1786	0.1585	0.52
Claude Sonnet 3.5	0.3675	0.3321	0.3490	0.49
DeepSeek-V3	0.4150	0.3750	0.3937	0.53
codellama-7b-hf	0.0632	0.6500	0.1153	0.23
codellama-13b-hf	0.0810	0.8786	0.1484	0.19
codellama-34b-hf	0.0723	0.7607	0.1321	0.25
starcoder2-15b	0.0742	0.3714	0.1237	0.14

5 Regression Bug Detection Evaluation (RQ3)

5.1 Overall Performance Analysis

Table 4 presents the comprehensive performance evaluation of 7 different large language models on the FSD task. DeepSeek-V3 achieved the highest overall performance with an F1 score of 0.3937, precision of 0.4150, and recall of 0.3750, demonstrating superior balance across all metrics. Claude Sonnet 3.5 followed closely with the second highest F1 score of 0.3490 and a precision of 0.3675, though its recall was slightly lower at 0.3321. GPT-4o performed relatively lower with an F1 score of 0.1585, precision of 0.1425, and recall of 0.1786. Although GPT-4o's precision was significantly lower compared to both DeepSeek-V3 and Claude Sonnet 3.5, it still remained substantially higher than the medium-to-small parameter models including the CodeLlama series and starcoder2.

The CodeLlama series and starcoder2 all exhibited precision below 0.1, but the CodeLlama models demonstrated high recall rates (0.6500-0.8786), significantly exceeding those of the large-parameter models. We observed that CodeLlama series models tend to make more positive predictions, resulting in higher false positive rates of approximately 0.9190-0.9368. Starcoder2 was relatively conservative (a recall of 0.3714, and a precision of 0.0742), but did not show outstanding performance in all metrics. In terms of explainability, DeepSeek-V3 achieved the highest score (0.53),

outperforming GPT-4o (0.52) and Claude Sonnet 3.5 (0.49). The medium- to small-parameter models all scored below 0.26, with starcoder2 achieving the lowest score (0.14).

Our further analysis reveals that LLMs' ability to explain compatibility errors is significantly weaker than that for intrinsic errors (around 8.20%), and the models' detection and explanation capabilities gradually deteriorate as the scale of change tokens increases. Detailed data are available on our site.

The experimental results indicate that large-parameter models demonstrate clear advantages in precision and explainability, with DeepSeek-V3 showing the most balanced performance across all evaluation metrics. Medium-to-small parameter models, despite detecting more function signatures, suffer from lower accuracy and higher false positive rates. Notably, DeepSeek-V3's superior performance across precision, recall, F1-score, and explainability establishes it as the most effective model for the FSD task among all evaluated approaches.

Finding 7: Compared to medium-to-small parameter models fine-tuned for code tasks, general large parameter models show greater potential in regression bug detection, but their capabilities are still limited. The best model achieves only a precision of 0.4105 and a recall of 0.3750.

5.2 Implications

Finding 7demonstrates that general large-parameter models can serve as a solid foundation for regression bug detection. However, these models still struggle to accurately explain compatibility errors unless they are provided with additional code components that may depend on the changed code. This requirement appears to conflict with the need to limit input size to avoid performance degradation. The commit decomposition strategy described in earlier finding 1 can be equally effective here, as we can reduce the number of change-related tokens while still including the auxiliary code context necessary for explaining compatibility errors.

6 An LLM-Based Tool Inspired by Our Findings

6.1 LLM4Reg

In this section, we present the design of LLM4Reg, a proof-of-concept tool inspired by our empirical findings. Algorithm 1 presents the workflow. Given a target commit for detection, LLM4Reg first uses SmartCommit to decompose the commit into multiple sub-commits (line 1), then applies the following pipeline to each sub-commit (line 3).

First, it uses [12] to extract the top-k methods with the highest change impact (lines 4-5) and the top-k frequently co-changed files (lines 6-7) to provide essential context for downstream prompt generation. We also employ SZZ [6] to supplement the co-change relationships between BICs and BFCs within the project. Subsequently, it constructs prompts for intrinsic error detection based on the code changes within the sub-commit and invokes the LLM to determine whether the sub-commit introduces intrinsic errors. The prompt for this phase is shown in Figure 4 (lines 8-9). If no intrinsic error is detected (line 10), the algorithm then generates compatibility error detection prompts based on the change impact

Algorithm 1: The Workflow of LLM4Reg

```
Input: commit C
Output: defect_results
// Phase 1: Commit decomposition
1  S ← DecomposeCommit(C);
2  defect_results ← ∅;
   // Phase 2: Analyze each sub-commit
3  foreach s ∈ S do
       // Change Impact Analysis
4      methods ← AnalyzeChangeImpact(s);
5      meths_topk ← GetTopKMethods(methods, k);
       // Co-frequency Change Analysis
6      F ← AnalyzeCoFrequencyChanges(s);
7      cochs_topk ← GetTopKCoChanges(F, k);
       // Intrinsic errors Detection
8      prompt_intr ←
         GenerateIntrErrorPrompt(s, meths_topk, cochs_topk);
9      result_intr ← LLMWorkflow(prompt_intr, "intr_err");
10     if result_intr.is_defect then
11         defect_results ←
             defect_results ∪ {("intr_err", result_intr)};
12     else
           // Compatibility Error Detection
13         prompt_comp ←
             GenerateCompPrompt(meths_topk, cochs_topk);
14         result_comp ←
             LLMWorkflow(prompt_comp, "comp_err");
15         if result_comp.is_defect then
16             defect_results ←
                 defect_results ∪ {("comp_err", result_comp)};
   // Return aggregated defect results
17 return defect_results;
```

methods and co-changed files, and calls the LLM again to identify compatibility error (lines 13-14). The prompt for this phase is shown in Figure 5. Here, we select DeepSeek as our LLM, which demonstrated the best performance in RQ3.

All detection results and explanations for sub-commits identified as buggy are integrated into the result set and returned as the algorithm's final output (line 17). In subsequent evaluation, a detection is considered correct if the result set contains actual regression bug detection results along with their explanations.

6.2 Evaluation

Setting. We evaluate LLM4Reg on FSD using the same evaluation methodology as RQ3, where DeepSeek-V3 is chosen as the baseline model. We also evaluate LLM4Reg against traditional Just-in-Time Defect Prediction (JIT-DP) methods (i.e., JITLine [38], LApredict [64], DeepJIT [16], Deeper [59], and CC2Vec [17]), which perform defect prediction also based on static analysis on code changes.

To analyze the contribution of each component, we conduct ablation experiments by disabling: the SmartCommit decomposition component (LLM4Reg$_{\neg sc}$), the workflow component that adopts different strategies based on root cause categories (LLM4Reg$_{\neg wf}$), or the co-occurring change component (LLM4Reg$_{\neg cc}$).

```
You are a code expert. Your task is to detect whether a
commit introduces regression bugs.

## Task: Intrinsic Errors Detection
Intrinsic errors are a common root cause of regression
bugs; they arise from errors within the code change
itself (e.g., logic errors, uncaught exceptions).

Please analyse the following code changes for any
intrinsic errors. Assume the surrounding method context
is correct.

<Diff>
```

Figure 4: Intrinsic Errors Detection Prompt

```
## Task: Compatibility Error Detection
The code change itself has been confirmed to be correct.

Based on the following information, determine whether
other code in the project must be adapted to these
changes.
Information:
- Top-k impacted methods:
<methods code>
- Top-k Co-changed Files:
<files code>
```

Figure 5: Compatibility Errors Detection Prompt

Table 5: The Performance of LLM4Reg and its Variants on the FSD Dataset Compared to the Baselines

Model/Method	Precision	Recall	**F1-score**	**Explain.**
DeepSeek-V3	0.4150	0.3750	0.3940	0.53
JITLine	0.6000	0.0107	0.0211	0.00
LApredict	0.1000	0.0071	0.0133	0.00
DeepJIT	0.1233	0.5393	0.2007	0.00
Deeper	0.1403	0.1964	0.1637	0.00
CC2Vec	0.1041	0.6571	0.1797	0.00
LLM4Reg	0.5400	0.4821	**0.5094**	**0.68**
LLM4Reg$_{\neg sc}$	0.4805	0.4393	0.4590	0.60
LLM4Reg$_{\neg wf}$	0.5111	0.4107	0.4554	0.57
LLM4Reg$_{\neg cc}$	0.5202	0.4607	0.4886	0.63

Results. Table 5 presents the evaluation results of our tool. Compared to the baseline model DeepSeek, LLM4Reg achieves significant performance improvements across all evaluation metrics. LLM4Reg improves precision by 0.125 (from 0.4150 to 0.5400), indicating that LLM4Reg produces fewer false positives and more accurately identifies actual regression bugs. The recall improvement of 0.1071 (from 0.375 to 0.4821) demonstrates LLM4Reg's enhanced ability to detect a higher proportion of actual defects in the dataset. The explanation score increases by 0.15 (from 0.53 to 0.68), showing that LLM4Reg not only detects defects more accurately but also provides more comprehensive and understandable explanations for the identified bugs.

As for JIT-DP methods, the performance results are lower than those of LLM4Reg in terms of F1-score. Moreover, these JIT-DP methods exhibit *zero* explanatory capability (Explain. = 0), indicating that they cannot explain the underlying root cause of the regressions, which is crucial for effective regression bug resolution.

The ablation study reveals the individual contribution of each component across the three evaluation dimensions. In terms of precision, disabling co-changes (LLM4Reg$_{\neg cc}$) maintains the highest precision (0.5202) among variants, while disabling commit decomposition (LLM4Reg$_{\neg sc}$) results in the lowest precision (0.4805), with a degradation of 0.0595 from the full model. For recall performance, removing the staged workflow (LLM4Reg$_{\neg wf}$) causes the most significant degradation, dropping to 0.4107 with a decrease of 0.0714 from full LLM4Reg, highlighting the importance of the two-stage detection approach for comprehensive defect identification. Regarding explanation quality, disabling the staged workflow (LLM4Reg$_{\neg wf}$) results in the poorest explanation scores (0.57), representing a decrease of 0.11 from the full model, suggesting that the structured workflow significantly contributes to generating more informative and accurate explanations for detected regression bugs.

7 Threats to Validity

7.1 Threats to Internal Validity

One internal threat is potential bias or error in data annotation. To mitigate this threat, we used a classification scheme derived from a vetted taxonomy [2] and applied an open coding scheme to incorporate new annotations as needed. We continuously assessed inter-rater agreement using Cohen's Kappa coefficient at each 10% completion interval, with ongoing discussions to clarify the annotation criteria and reconcile inconsistencies overseen by a senior supervisor. These adjustments led to a stable, high agreement level (Kappa > 0.83), demonstrating rigorous control over annotation quality and minimizing potential biases in our results.

Another internal threat relates to potential issues in the tools and feature extraction scripts used for evaluating defect detection on regression bugs. To mitigate this threat, we sourced the tools and scripts directly from GitHub, using the same hyperparameters specified in the original studies. The authors also carefully verified the correctness of the code and the accuracy of the extracted features based on existing literature.

7.2 Threats to External Validity

One external threat is the diversity and representativeness of the regression bugs in our dataset, specifically whether our findings can be generalized to a broader range of regression bugs. To address this, we sampled from the largest available Java regression bug dataset, which contains 1,025 bugs from 147 different open-source projects and has been shown in prior work [41] to offer greater diversity than Defects4J [20]. We also performed sampling with a high confidence level and margin.

Another external threat concerns the representativeness of the LLMs selected for evaluation. We selected three large-parameter models that are commonly used for code tasks. Additionally, we included medium and small-parameter models that have been fine-tuned specifically for code tasks, to assess the performance differences across models of varying sizes.

8 Related Work

8.1 Regression Bug Analysis

There has been a series of work that attempted to consolidate comprehensive understanding of regression bugs. Nir et al.[36] found that bug fixes often introduce regression bugs and proposed a tool named CodePsychologist to help programmers locate the code segments causing these regression bugs. Khattar et al.[24] investigated regression bugs, reported that more than half of them should be assigned a high priority, and identified code changes responsible for introducing regression bugs in Google Chromium project. Xiao et al. [57] showed that approximately half of Linux bugs could be classified as regression bugs. In a subsequent study, Xiao et al. [56] empirically analyzed regression bug chains in Linux, modeling the relationships between regression bugs and commits as a directed bipartite network.

8.2 LLM-based Defects Detection

LLM-based defect detection has recently attracted growing interest due to LLMs' remarkable capabilities in code understanding and generation. Early studies in this area primarily focused on leveraging LLMs to assist static analysis tools by filtering out false positives, thereby improving the practical usability of these tools [23, 51]. Subsequent works have proposed more direct integration of LLMs into the defect detection pipeline. For example, LLift [29] combines path-sensitive static analysis with LLM reasoning to better identify real bugs in complex codebases like the Linux kernel. Meanwhile, other efforts have explored using LLMs for vulnerability and bug detection through enhanced code representation, such as incorporating graph structural information and in-context learning to boost LLM effectiveness on public vulnerability datasets [31]. In addition to detection, LLMs have also been used for automated vulnerability repair and patch porting [55]. Despite these advances, recent systematic evaluations [46] highlight significant challenges: current LLMs often exhibit unstable performance, lack robustness to minor code changes, and may fail to provide reliable reasoning in real-world scenarios. While LLM-based approaches are closely related to our work, we have not directly applied them in this paper, as existing studies typically focus on general or security-related defects and do not specifically target regression bugs.

8.3 Defect Prediction in Code Changes

Just-in-Time Defect Prediction (JIT-DP) aims to predict potential defects at the moment of source code changes. These works are closely related to our regression bug change identification. Early works [21, 26, 34] focused on extracting structured features from code changes in commits to analyze the risk of defects being introduced. Later works[8, 10, 30, 58, 59] primarily revolved around the 14 expert features and attempted different machine learning and deep learning models, such as convolutional neural networks, decision trees, and others. Then DeepJIT [16], CC2Vec [17], and JITLine [38] leverage neural networks to automatically learn representations directly from code changes, building upon manually crafted expert features. JITLine additionally employs LIME[39] to provide explanations for model predictions, thereby enhancing interpretability. More recently, JIT-Smart [9] and JIT-Fine [35] integrate code semantic features extracted by the pretrained language model CodeBERT [11] with expert features, enabling both defect prediction and localization of the specific changes that introduce defects. While these two approaches align more closely with our research objectives, we do not include them in our comparative evaluation because there currently exists no large-scale regression bug dataset with fine-grained annotations at the level of individual bug-inducing changes, which precludes the direct application of these models to regression bug prediction.

9 Conclusions and Future Work

In this study, we present the first comprehensive analysis of the root causes and development activities of regression bugs, as well as the knowledge required to understand them, based on a dataset of 280 regression bugs from 84 open-source projects. We further evaluated the performance of SOTA LLMs in detecting these real-world regression bugs, revealing their current limitations. Based on our findings, we propose LLM4Reg, which significantly improves the effectiveness of LLMs for regression bug detection. While the performance metrics of our tool is far from satisfactory, we believe it is a demonstration of a promising and evidence-backed step forward.

In the future, we will focus on extracting the expected behaviors of features from their historical changes, and explore how LLMs can be further leveraged to supplement the semantic relationships between code segments. These efforts aim to enable more effective detection of logical error and compatibility error for regression bug. We also acknowledge the importance of generalizing our findings to other languages. We plan to extend our work to other languages, such as Python, to assess broader applicability of our approach.

10 Data Availability

The dataset is available in our Github repository https://github.com/FudanSELab/LLM4Reg.

Acknowledgments

The authors sincerely thank anonymous reviewers for their valuable comments and helpful suggestions. This work was supported by National Key R&D Program of China (2023YFB4503805) and National Natural Science Foundation of China (62172099).

References

[1] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 104–113.

[2] Boris Beizer. 1984. *Software system testing and quality assurance*. Van Nostrand Reinhold Co.

[3] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1–12.

[4] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *ESEC/FSE 2013*. https://api.semanticscholar.org/CorpusID:887319

[5] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: studying complexity of regression errors. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 105–115. doi:10.1145/2610384.2628058

[6] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 27, 2019*, Francesca Arcelli Fontana, Bartosz Walter, Apostolos Ampatzoglou, Fabio Palomba, Gilles Perrouin, Mathieu Acher, Maxime Cordy, and Xavier Devroey (Eds.). ACM, 7–12. doi:10.1145/3340482.3342742

[7] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 1–5.

[8] George G. Cabral, Leandro L. Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 666–676. doi:10.1109/ICSE.2019.00076

[9] Xiangping Chen, Furen Xu, Yuan Huang, Neng Zhang, and Zibin Zheng. 2024. JIT-Smart: A Multi-task Learning Framework for Just-in-Time Defect Prediction and Localization. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1–23. doi:10.1145/3643727

[10] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Inf. Softw. Technol.* 93 (2018), 1–13. doi:10.1016/J.INFSOF.2017.08.004

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139

[12] Ismail Sergen Göçmen, Ahmed Salih Cezayir, and Eray Tüzün. 2025. Enhanced code reviews using pull request based change impact analysis. *Empir. Softw. Eng.* 30, 3 (2025), 64. doi:10.1007/S10664-024-10600-2

[13] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.

[14] Claire Le Goues, Thanhvu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72. https://api.semanticscholar.org/CorpusID:4111307

[15] P. Henry. 2008. The testing network: An integral approach to test activities in large software projects. *tThe Testing Network: An Integral Approach To Test Activities In Large Software Projects* (01 2008), 1–438. doi:10.1007/978-3-540-78504-0

[16] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 34–45. doi:10.1109/MSR.2019.00016

[17] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: distributed representations of code changes. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 518–529. doi:10.1145/3377811.3380361

[18] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 31–37.

[19] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse hierarchical delta debugging. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 194–203.

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[21] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.* 39, 6 (2013), 757–773. doi:10.1109/TSE.2012.70

[22] Alireza Khalilian, Ahmad Baraani-Dastjerdi, and Bahman Zamani. 2021. CGen-Prog: Adaptation of cartesian genetic programming with migration and opposite guesses for automatic repair of software regression faults. *Expert Syst. Appl.* 169 (2021), 114503. https://api.semanticscholar.org/CorpusID:232022450

[23] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin B. Clement, and Neel Sundaresan. 2022. Learning to Reduce False Positives in Analytic Bug Detectors. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1307–1316. doi:10.1145/3510003.3510153

[24] Manisha Khattar, Yash Lamba, and Ashish Sureka. 2015. SARATHI: Characterization Study on Regression Bugs and Identification of Regression Bug Inducing Changes: A Case-Study on Google Chromium Project. *Proceedings of the 8th India Software Engineering Conference* (2015). https://api.semanticscholar.org/CorpusID:17833019

[25] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 16–22.

[26] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin 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 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 111–120. doi:10.1109/ICSM.2015.7332457

[27] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 821–830.

[28] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 474–499. doi:10.1145/3649828

[29] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 474–499. doi:10.1145/3649828

[30] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.). IEEE Computer Society, 11–19. doi:10.1109/ESEM.2017.8

[31] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *J. Syst. Softw.* 212 (2024), 112031. doi:10.1016/J.JSS.2024.112031

[32] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[33] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[34] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Tech. J.* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[35] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 672–683. doi:10.1145/3540250.3549165

[36] Dor Nir, Shmuel S. Tyszberowicz, and Amiram Yehudai. 2007. Locating Regression Bugs. In *Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4899)*, Karen Yorav (Ed.). Springer, 218–234. doi:10.1007/978-3-540-77966-7_18

[37] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Softw. Pract. Exper.* 46, 9 (Sept. 2016), 1155–1179.

[38] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 369–379. doi:10.1109/MSR52588.2021.00049

[39] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1144. doi:10.1145/2939672.2939778

[40] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: a graph-based interactive assistant for activity-oriented commits. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 379–390. doi:10.1145/3468264.3468551

[41] Xuezhi Song, Yun Lin, Siang Hwee Ng, Yijian Wu, Xin Peng, Jin Song Dong, and Hong Mei. 2022. RegMiner: towards constructing a large regression dataset from code evolution history. In *ISSTA '22: 31st ACM SIGSOFT International Symposium*

on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 314–326. doi:10.1145/3533767.3534224

[42] Xuezhi Song, Yijian Wu, Shuning Liu, Bihuan Chen, Yun Lin, and Xin Peng. 2024. C2D2: Extracting Critical Changes for Real-World Bugs with Dependency-Sensitive Delta Debugging. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 300–312. doi:10.1145/3650212.3652129

[43] Hamed Taherdoost. 2017. Determining sample size; how to calculate survey sample size. *International Journal of Economics and Management Systems* 2 (2017).

[44] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 1 (2015), 471–482. https://api.semanticscholar.org/CorpusID:17125466

[45] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 862–880. doi:10.1109/SP54263.2024.00210

[46] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 862–880. doi:10.1109/SP54263.2024.00210

[47] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[48] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 881–892.

[49] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* (2019).

[50] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* 18, 7 (2024), 168. doi:10.1145/3653718

[51] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* 18, 7 (2024), 168. doi:10.1145/3653718

[52] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing Chi Cheung. 2019. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering* 47 (2019), 2348–2368. https://api.semanticscholar.org/CorpusID:207784361

[53] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 326–337. doi:10.1145/3338906.3338962

[54] W. Eric Wong, Joseph Robert Horgan, Saul London, and Hiralal Agrawal. 1997. A study of effective regression testing in practice. *Proceedings The Eighth International Symposium on Software Reliability Engineering* (1997), 264–274. https://api.semanticscholar.org/CorpusID:2911517

[55] Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Chen, Zhuotong Zhou, Yiheng Huang, JunPeng Zhao, and Xin Peng. 2025. Mystique: Automated Vulnerability Patch Porting with Semantic and Syntactic-Enhanced LLM. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE007 (June 2025), 23 pages. doi:10.1145/3715718

[56] Guanping Xiao, Zheng Zheng, Bo Jiang, and Yulei Sui. 2020. An Empirical Study of Regression Bug Chains in Linux. *IEEE Transactions on Reliability* 69 (2020), 558–570. https://api.semanticscholar.org/CorpusID:198462273

[57] Guanping Xiao, Zheng Zheng, Beibei Yin, Kishor S. Trivedi, Xiaoting Du, and Kai-Yuan Cai. 2017. Experience Report: Fault Triggers in Linux Operating System: from Evolution Perspective. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*. IEEE Computer Society, 101–111. doi:10.1109/ISSRE.2017.21

[58] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Inf. Softw. Technol.* 87 (2017), 206–220. doi:10.1016/J.INFSOF.2017.03.007

[59] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*. IEEE, 17–26. doi:10.1109/QRS.2015.14

[60] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. 2015. A synergistic analysis method for explaining failed regression tests. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 257–267.

[61] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. 26–36.

[62] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[63] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes* 27, 6 (2002), 1–10.

[64] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 427–438. doi:10.1145/3460319.3464819