Yet another trace-based approach for the cause of software regressions in JavaScript and Python

Feng Dai[1,*], Yuefeng Hu[1], Tetsuro Yamazaki[1], and Shigeru Chiba[1]
[1]The University of Tokyo, Bunkyo, Tokyo, Japan
daifeng@csg.ci.i.u-tokyo.ac.jp, huyuefeng99@csg.ci.i.u-tokyo.ac.jp, yamazaki@csg.ci.i.u-tokyo.ac.jp, chiba@acm.org
*corresponding author

Abstract—Addressing the cause of software regressions is an important but difficult task, and has not been well studied. Current tools have some limitations, such as low detection accuracy. In this paper, we try to address these limitations and improve the accuracy of locating causes of software regressions by proposing three new techniques. Our techniques are based on tracing source code changes during program execution. Moreover, we extend current benchmark to a new programming language, and compare our techniques against existing techniques on the extended benchmark. Our new techniques outperform existing techniques in accurately locating the root cause of software regressions. Specifically, our LLM-powered technique achieves the state-of-the-art accuracy.

Keywords–Software Regression; Debugging; Code Tracing; Software Engineering; Large Language Models

1. Introduction

Locating the root cause of software regressions, where a feature that has worked before stops functioning properly after updates, is important but difficult [1], [2]. Usually, developers have to manually check function calls based on testing outputs and revision histories. Such debugging is especially tedious when the project is large and different components are interconnected with each other. A recent work [3] provides an automatic tool to localize the cause of software regressions based on runtime tracing. They analyze dynamic information such as inputs and outputs of function executions and find discrepancies between a base program and an updated but faulty program. Their attempt to trace dynamic information during program execution is successful, but still has some limitations. First, the accuracy of their approach is less satisfying. Users still have to manually check multiple candidates to locate the true cause. Second, their approach is limited to projects written by JavaScript. It is unknown whether this approach can be applied to other languages.

In this paper, we extend the work by [3] to address the previously mentioned limitations. First, we provide three new techniques to improve the retrieval accuracy of the root causes of software regressions, including two heuristic algorithms and one large language model (LLM)-powered technique. In specific, our techniques also dynamically trace program executions between the base program and the faulty program. However, we utilize only the order of function executions instead of input/output information, which is easy to implement. Moreover, our LLM-powered technique achieves the state-of-the-art accuracy compared with other heuristic algorithms.

Second, we also extend current benchmark and support another programming language, Python. In detail, we extract 12 real-world Python regressions from the BugsInPy [4] benchmark, and implement a Python-based tracer both for our algorithms and for algorithms introduced by [3]. Along with 11 JavaScript regressions introduced by [3], we conduct a detailed analysis to locate the cause of software regressions for both Python and JavaScript projects.

Our contributions are two-fold:

1) We create a new dataset containing software regressions written in Python. This is aimed for an extensive evaluation for tools that try to detect the cause of software regressions. All regressions in the dataset are based on open-source software projects and are suitable for real-world analysis. We make our dataset public for future analysis[1].
2) We improve the accuracy of existing approaches by providing three new techniques. We validate our techniques on both Python and JavaScript benchmark. Among these techniques, the one that utilizes a large language model achieves the best accuracy. We make our source code public for further research[2].

The rest of the paper is organized as follows. We discuss the motivation of our work in Section 2. Section 3 presents the details of our idea. Section 4 presents the results and analysis of our experiments and Section 5 presents our discussion. Finally, we present related works in Section 6 and conclude this paper in Section 7.

2. Retrieve the Cause of Software Regressions

While many works have focused on detecting software regressions or preventing regressions before release, locating the cause of a software regression is still difficult and needs exploration. In Figure 1, we show an example of a real-world regression from a JavaScript project called Express [5], which is a web framework for Node.js providing a set of features for web and mobile applications. This example involves the version with commit id 997a558 (referred as the base version) and its subsequent commit with id c6e6203 (referred as the

[1]https://zenodo.org/records/15203310
[2]https://github.com/Eduardo95/bugbee

faulty version). In this example, the green code belongs to the base version and the red code belongs to the faulty version.
In this example, the faulty version implicitly introduces a regression by mistakenly encoding the uniform resource identifier (URI) from the arguments provided in the function. As shown in Figure 1, line 3 from the faulty version assigns the argument to a variable *address*. Then the *if* statements check whether there are two arguments and if the first argument is a number. If yes, the first argument is assigned to *status* as status code and the second argument is assigned to *address* as the url address. If there is only one argument, the code would not enter these conditional branches. However, later at line 20, the faulty version does not use the parsed *address* but the original argument *url* to encode the uniform resource identifier, which makes the URI incorrect. To side with the modifications in Figure 1, the developers also change the corresponding test case in Figure 2 by deleting the status code in line 7. Along with many other changes, the developers commit this change without carefully reviewing the code. This bug is identified as a regression later and gets fixed by developers at a later commit (6f91416).

```
1  res.redirect = function(url) {
2  - var head = 'HEAD' == this.req.method;
3  + var address = url;
4    ...
5  - if (2 == arguments.length) {
6  -   if ('number' == typeof url) {
7  -     status = url;
8  -     url = arguments[1];
9  + if (arguments.length === 2) {
10 +   if (typeof arguments[0] === 'number')
       {
11 +     status = arguments[0];
12 +     address = arguments[1];
13     } else {
14       ...
15     }
16   }
17
18   this.format({
19     text: function(){
20       body = statusCodes[status] + '.
             Redirecting to ' + encodeURI(url
             );
21     },
22     ...
23   });
24
25 - ...
26 + ...
27 }
```

Figure 1: Base (green) and updated (red) faulty version of function *res.redirect* in response.js.

In most cases, addressing the root cause of software regressions requires manually checking source code, and can be burdensome. Sometimes, even if a programmer knows which unit test fails, it is still tedious. In the given example, *res.redirect* is just a middle-ware function in *response.js* to handle web requests, while *response.js* also relates to many other functions. All these functions could be a potential candidate of the root cause of the software regression when developers are given limited information from the failures of unit tests by testing frameworks such as Mocha [6]. Moreover, the regression is usually committed with many other changes. In Table I, we present an analysis of code changes between a base version and a updated but faulty version, which is collected from real-world regressions. It can be seen that in average, 7 files are changed for Python projects and 5 files are changed for JavaScript projects when regressions happen, and more than 300 lines of code are changed for both languages. It is difficult for developers to locate the root cause of a software regression among these changes manually. The table shows the number of changed files and lines of code between two consecutive commits. It is even harder if the regression is discovered after a long period of time along with numerous commits covering the regression. Given the complexity of pinpointing software regressions, an automated tool would become greatly convenient and time-saving for developers to resolve the task efficiently.

A recent work by [3] presents a trace-based tool to locate the root cause of software regressions automatically. Their main idea is to trace the inputs and outputs of every function invoke during the execution of both base and faulty program when a regression happens. In specific, they insert tracing statements in both base and faulty version of source code to record function behaviour during program execution. Based on the dynamic behaviour, they construct a data structure called function call graph, and compare the differences between call graphs from both base version and faulty version. Based on the differences, two heuristic strategies are used to locate the root function that may cause the occurrence of a software regression.

This work demonstrates that detecting causes of software regressions by tracing software execution is possible. However, there are still some limitations about their algorithms. First, the accuracy of their algorithms is not high. Each of their heuristics discovers only around half of real causes of regressions

```
1  it('should include the redirect type',
       function(done){
2    ...
3  - app.use(function(req, res){
4  -   res.redirect(301,
       'http://google.com');
5  - });
6  + app.use(function(req, res){
7  +   res.redirect('http://google.com');
8  + });
9    ...
10
11 })
```

Figure 2: Base (green) and updated (red) faulty version of test case related to function *res.redirect*.

TABLE I: Changes between commits for software regressions

Project Name	Avg. files	Avg. lines of code involved
Regressions in youtube-dl (Python)	3	31
Regressions in Tornado (Python)	20	1420
Regressions in black (Python)	2	70
All regressions in Python projects	7	384
Regressions in Express (JavaScript)	4	188
Regressions in ESLint (JavaScript)	6	655
Regressions in hessian.js (JavaScript)	7	134
All regressions of JavaScript projects	5	305

even by giving many candidates. This still requires developers to manually check these candidates to locate the true cause, which is trivial and time-consuming. Second, their approach only covers projects written by JavaScript and heavily depends on the serialization feature of JavaScript. It might not be easy to implement such a tracer for other languages, especially on top of the off-the-shelf engines. It is unknown how their algorithms perform when applied to projects written in other programming languages.

3. Proposal

We extend the approach proposed by [3] to address the limitations mentioned in Section 2. We first generate execution traces of both the base program and the faulty program, and then compare the execution traces. Different from collecting inputs and outputs of functions, we only need the function execution order of each program, and we analyze the source code of these functions. Based on the hypothesis that the cause of a software regression could be located in the differences between the base program and the faulty program, we present three techniques to improve the accuracy against the algorithms in [3]. We implement a program tracer different from the one presented by [3] and we open-source our code in GitHub[3]. Our implementation is easier and more friendly to many off-the-shelf engines.

The system overview is shown in Figure 3. Our approach depends on a version control system, such as Git, to maintain the version of source code. In our scenario, a software regression happens when a previous version works properly with a given unit test but a later updated version displays some buggy behaviour with the same unit test. We refer the previous version as the base version and the later version as the faulty version. Our approach tries to resolve the cause of such regressions. To do so, we will introduce our tracer and our techniques to retrieve the cause of software regressions in the following sections.

[3]https://github.com/Eduardo95/bugbee

3.1 Trace execution order of functions

We trace the execution order of functions as dynamic information for analysis. In our approach, we trace the function execution order of the base version and the faulty version seperately. We will obtain a tree structure that represents the invoking relationships between functions in the program. In this tree, a parent node is a function to invoke others and a child node is a function being invoked by its parent within the function body. Specifically, the root node indicates the entry point of a program, which is artificially constructed. An example of the execution trace is shown in Figure 4. In this example, the program executes function A, B and D sequentially. Before the program executes function B, function A will invoke function B and C first. To note, the same function can be invoked multiple times by the same or by different functions.

The tracing is done by modifying the functions and inserting tracing statements into the original source code. These statements will record function order during program execution. Moreover, when modifying the functions, the original source code of each function is recorded for further comparison. In our experiments, we only modify the source code of each project itself, without considering the libraries it depends on. An example of the modification is shown in Figure 5. The original function is wrapped and bound to its context so that its behavior remains unaffected. We do some operations before and after its execution in order to construct a tree-like execution structure. Specifically, Python does not have a *this* context like JavaScript; instead, it uses *self* to refer to the instance within class methods. We consider such things when modifying the source code.

Our tracer supports various types of functions in a program, including module-level functions, methods, lambda functions in Python, function expressions and arrow functions in JavaScript, and so on. A unique identifier is given to each function to represent this function. This identifier includes the path of the source file, the line number in the source file, the name of the function if any, and hash value of the function definition. The hash value is calculated after pretty-printing to avoid the influence of formatting. An example is shown below:

```
tornado/options.py@line99/Func@__init__,75f93a90c.
```

It shows that the function is defined in line 99 of the file *tornado/options.py*. It is a module-level function and its function name is *__init__*. The hash value of this function is *75f93a90c*.

3.2 Compare execution traces

We propose three techniques to compare two execution traces and retrieve the cause of software regressions. Two of them are based on heuristic algorithms and the other one is powered by a large language model (LLM).

Our heuristic algorithms are based on the assumption that a regression is usually caused by the changes made in a certain function. We want to locate such a change that causes the regression among various changes in various files. In order to compare two different but homologous execution traces, we

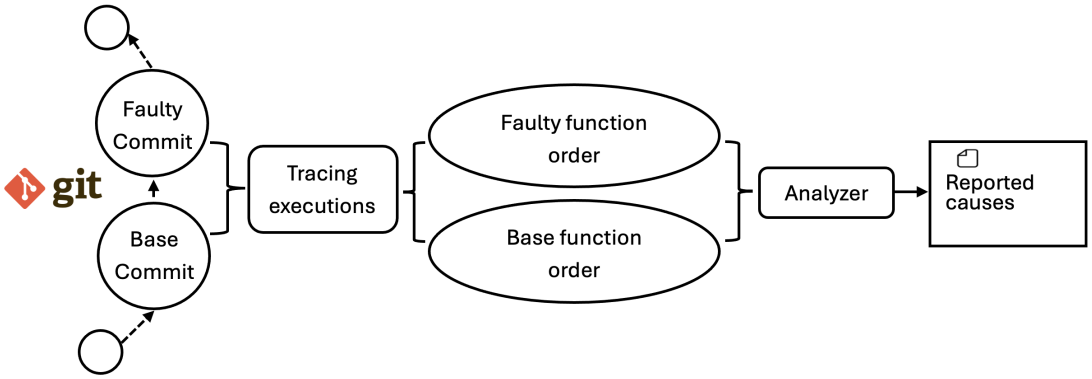


Figure 3: The system overview.

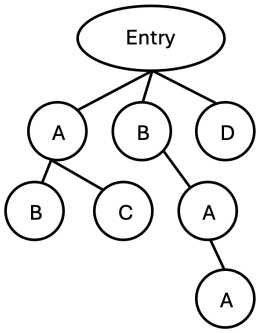


Figure 4: An example of the execution trace.

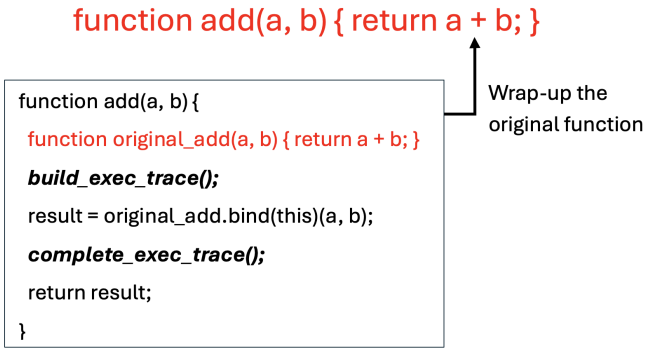
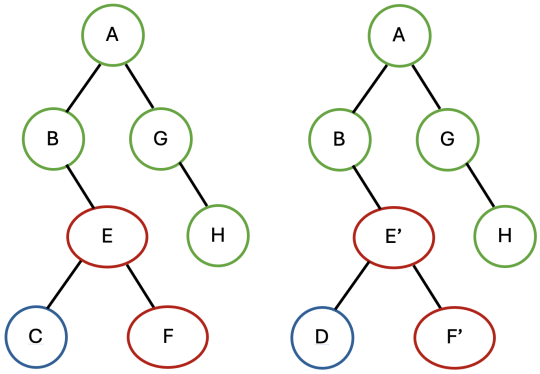


Figure 5: An example of tracing statement insertion.



(a) The base version. (b) The faulty version.

Figure 6: A comparison between two callgraphs

set up two rules to avoid the influence of irrelevant functions. First, only a pair of nodes with the same position in traces can be compared. The same position means they have the same depth and breadth. Second, only a pair of nodes with the same function type and name can be compared. These two rules guarantee that a pair of nodes are always following the same execution logic, despite the fact that they may have different behaviour. We compare each pair of nodes in depth-first order. We give an example in Figure 6. In this example, the blue node C on the left means a function with a name C, and the blue node D on the right means a function with a name D. In this case, they are at the same position, but have different function names. Therefore, they are not compared although they are at the same position. The red nodes E and E' mean that they have the same function name E, but the source code is changed. The red nodes F and F' also have the same name F but changed source code. In this case, both function pairs, E/E' and F/F', are compared as they are at the same position and have the same name.

The other technique utilizes the power of LLMs to understand source code. Recently, LLMs have demonstrated abilities in processing source code and finishing code-related tasks, such as code completion [7]. Given two execution traces and the source code, we want to know whether LLMs can understand the logic of the programs and pinpoint the function that introduces faulty behaviour. To this extend, we design the third technique as a comparison. We will introduce our techniques in detail.

a) Deepest change with the same position: The first heuristic technique tries to find the deepest pair of functions in execution traces that have a change in the source code. The deepest pair means that they have the deepest depth compared with other different pairs, and they do not have descendants that are different in their subtrees. If two pair has the same depth, the first pair is reported. As shown in Figure 6, function F and F' display discrepancy in source code and are the deepest

in execution traces. This pair is reported by this technique. The reasoning behind this technique is that the deepest pair contains no further deviated behaviour in their nested calls to other functions, and they are responsible for the faulty behaviour between the base version and the faulty version.

An example is shown in Figure 7. In this example, the *create()* function from *rule-tester.js* calls the *create()* function from *no-multi-spaces.js*, and the *create()* function from *no-multi-spaces.js* calls the *Program()* function which is defined inside the *create()* function from *no-multi-spaces.js*. As the source code of the *Program()* changes, the source code of the *create()* function from *no-multi-spaces.js* changes correspondingly. By the deepest-change algorithm, the *Program()* function is considered as the cause of this regression. This is a real example from the project ESLint with bug id 307. According to the results in section 4, the deepest-change algorithm reports a correct cause of the regression.

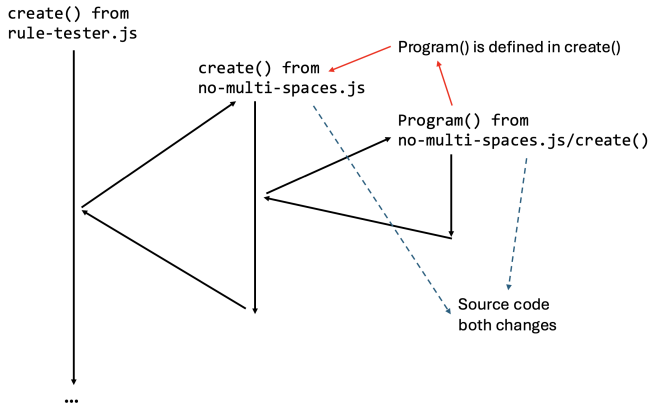


Figure 7: Deepest change with the same position.

b) First change with the same position: The second heuristic technique tries to find the first pair of functions in execution traces that has a change. The first pair means that they appear first in the order of depth-first search following the previous rules. As shown in Figure 6, function E and E' are the first pair with the same name and the same position, but are different in source code. This pair is reported. The reasoning behind this technique is that the first chronologically different pair identifies the beginning of deviated behaviour between the base version and the faulty version and is the root cause of the bug.

In the example shown in Figure 7, the first-change algorithm should report the *create()* function from *no-multi-spaces.js* as the cause of the regression. However, this is not a correct report.

c) An LLM-powered technique: The third technique utilizes an LLM to locate the function that causes a regression. It relies on the power of LLMs to understand source code. We want to know whether an LLM can correctly pinpoint the cause of a regression if it is provided with the source code, the order of functions executed and probably some other information. However, due to context limits of LLMs, it is not possible to input source code of the whole project into an LLM. Therefore, we use a sliding window through the linearized execution trace to reduce the context.

The basic idea is to linearize the execution trace by depth-first-search order, and extract a sub-trace from the linearized trace to give to the LLM. As shown in Figure 8, the execution traces in Figure 6 are linearized by depth-first-search order. To extract a sub-trace, we compare the functions in two linearized traces one-by-one to find the first pair of functions that are different. This time, the pair of functions does not have to have the same name or the same position in the original tree structures. The two linearized execution traces start to deviate by such the first pair of different functions. After getting such a pair, we expand the pair into a sub-execution trace by including some pairs before and after this pair. This is for an LLM to better understand the context of function executions. In this example, the rectangle with dotted lines is the sub-execution traces we obtain. The sub-execution traces are given to an LLM to locate the cause of the regression. The window size is adjustable depending on the input size of the LLM. Also, it is possible that the root cause of a regression is not contained in the sub-execution trace. In such cases, the LLM is not able to locate the root cause of a regression. A large window size reduces the chance of occurrences of such cases. In our experiments, we include five functions before the starting point and twenty functions after the starting point.

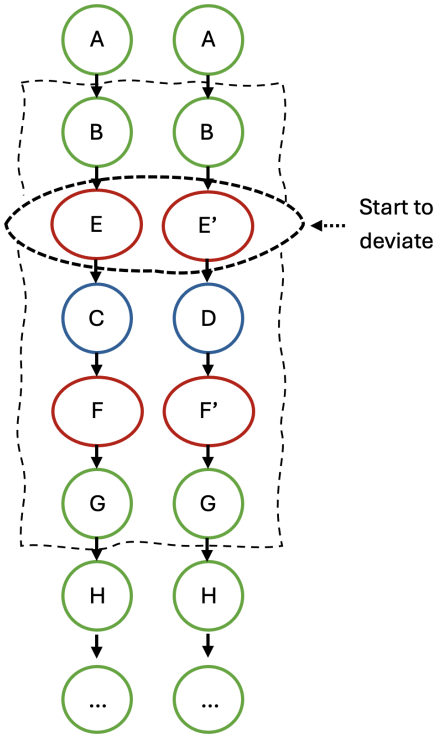


Figure 8: A sliding window through the linearized execution trace.

Moreover, to help an LLM better analyze the source code, we provide the error messages of the unit test from the faulty

version as an enhancement. In real development, developers usually need error messages to help them better locate where the bug happens. We wonder whether such messages can improve the accuracy of an LLM as they usually help developers. To find out, we do both experiments of using an LLM with and without the error messages. We provide the prompt of using an LLM with error messages in Figure 9a. As a comparison, we also provide the prompt of using an LLM without error messages in Figure 9b for experimental purpose.

```
You will be given a list of function
call pairs and an error message
{error_messages}. In each pair, one
is a function call from the correct
version, and the other one is a
function call from the faulty version.
The list is in the order of function
calls. Based on the list and the error
message, please tell me which function
might be the cause of the regression
bug. The list is
{str(list_of_function_pairs)}.
Please be as precise as possible, and
just give me the function name.
```

(a) Prompts with error messages

```
You will be given a list of function
call pairs. In each pair, one is a
function call from the correct version,
and the other one is a function call
from the faulty version. The list is
in the order of function calls. Based
on the list, please tell me which
function might be the cause of the
regression bug.
The list is
{str(list_of_function_pairs)}.
Please be as precise as possible, and
just give me the function name.
```

(b) Prompt without error messages as a comparison

Figure 9: An example of prompts in finding the cause of software regressions

4. EXPERIMENTS

In this paper, we evaluate the accuracy and overhead of locating the cause of software regressions of our techniques against the algorithms mentioned in [3]. The authors mention two algorithms, first-deepest-function (FDF) algorithm and Top-n algorithm. First deepest function algorithm identifies the starting point of deviated behaviour of the faulty version. However, they consider inputs and outputs of functions instead of source code. Top-n algorithm counts the occurrences of each function that has different behaviour, and selects the most-occurred n functions. This algorithm requires users to spend more time investigating the candidates and determine the true cause. In this paper, we choose 2 as the number of n as we think that such kind of algorithms should not provide many candidates for users to choose to avoid false-positive issue. Since [3] does not provide a Python version tracer, we implement a Python tracer to trace inputs/outputs based on the description and source code provided in the paper.

In this section, we will introduce our experiment settings, datasets and experiment results.

4.1 Experiment settings

The experiments are conducted on a machine with an *11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz* CPU and 32GB RAM, and the operating system is Ubuntu 20.04.4 LTS. The versions of Node.js and Python we use vary according to the requirements of each project of each respective regression. Our third technique utilizes a large language model for analysis. In this paper, we use *gpt-4-turbo* API from OpenAI as our backend model as it is one of the most easily-available and state-of-the-art models on the market.

4.2 Datasets

Currently, there is no dataset with software regressions covering multiple programming languages. The authors in [3] construct a dataset with 12 regressions written in JavaScript based on an existing dataset BugsJS [8], which contains over 450 manually-validated JavaScript bugs from 10 projects. The authors manually screen and extract 12 regression bugs. In this paper, we reuse the dataset but remove one regression from the original dataset (The bug in Express with Bug-id 1). The reason of removal is that we find that this regressions is not caused by the function provided in the dataset. In fact, the causing function provided in the dataset is never invoked in both base version and faulty version of this regression. In addition, we re-check all other JavaScript bugs and make sure no further regressions can be found in the dataset.

Moreover, we extend the existing dataset to cover a new programming language, Python. We extract regressions based on an existing Python bug dataset BugsInPy [4], which contains over 400 real-world Python bugs from 17 Python projects. The approach we adopt to extract regressions is as follows. In detail, each bug in BugsInPy corresponds to a real Git commit, a unit test and a fix. Starting from the commit of the bug, we traverse the Git tree in reverse chronological order and execute the fixed unit test on all preceding commits. All preceding commits should report faulty. If we ever get a test success after a test failure in reverse chronological order, it could be considered a regression as the same functionality works properly before it stops working. The commit with the successful test result is the base version, while the commit with the failed test result is the faulty version. The function being updated in a later fix is identified as the cause of the regression. We manually verify the extracted regressions again to ensure the validity of the regressions, along with their corresponding unit tests and fixes.

We manage to extract 12 regressions from three projects from BugsInPy. We collect the commits of the base and faulty version, along with their fixes. In detail, 7 regressions are from a project called youtube-dl, 3 regressions are from a project called tornado and 2 regressions are from a project called black. We also collect the error messages from the faulty

version as supporting materials, including the regressions from JavaScript projects. An example of an error message is shown in Figure 10. In total, we have 23 real-world software regressions from two different programming languages. Our dataset is publicly available Zenodo[4].

The number of regressions is relatively small. This is because that in a typical software development process, a software regression should be addressed before it is committed. In other words, software regressions should not appear in the version control system such as Git. Therefore, it is highly challenging to identify software regressions in publicly available Git repositories. However, this does not mean that our research topic is less important. In fact, our tool is designed to help developers fix such regressions before they are committed to the repository, which improves the quality of the software development process. Based on this reason, we construct a dataset covering two programming languages to test the validity of our approach. Some details of these regressions are summarized in Table II. The table shows the number of changed files and lines of code between two consecutive commits. Considering that regressions are often discovered several commits after they are introduced, the actual numbers are likely even higher.

```
1) app should support empty string path:
  TypeError: Router.use() requires callback function but
      got a [object String]
    at /.../lib/router/index.js:389:17
    at /.../lib/router/index.js:405:113
    at Array.forEach (<anonymous>)
    at Function.INNERFUNC (/.../lib/router/index.js:384:15)
    at Function.use (/.../lib/router/index.js:418:45)
    at /.../lib/application.js:174:25
    at /.../lib/application.js:218:113
    at Array.forEach (<anonymous>)
    at Function.INNERFUNC (/.../lib/application.js:171:9)
    at Function.use (/.../lib/application.js:231:45)
    at context.Original (/.../test/app.use.js:10:13)
    at context.<anonymous> (/.../test/app.use.js:32:113)
    at Runnable.run (/.../node_modules/mocha/lib/runnable.
        js:217:15)
    at Runner.runTest (/.../node_modules/mocha/lib/runner.
        js:373:10)
    at /.../node_modules/mocha/lib/runner.js:451:12
    at next (/.../node_modules/mocha/lib/runner.js:298:14)
    at /.../node_modules/mocha/lib/runner.js:308:7
    at next (/.../node_modules/mocha/lib/runner.js:246:23)
    at Immediate.<anonymous> (/.../node_modules/mocha/lib/
        runner.js:275:5)
    at process.processImmediate (node:internal/timers
        :476:21)
```

Figure 10: An example of an error message

4.3 Results

We present the experiment results in Table III and Table IV. From the results, we successfully locate more causes of regressions against the algorithms that trace I/O from [3]. For each regression, we demonstrate whether each technique can successfully locate the cause of the regression. In Python experiments, both our first-change algorithm and the LLM-powered technique with error messages achieve the best accuracy against other techniques. Our deepest-change algorithm can resolve 7 regressions, and first-change algorithm can resolve 9 regressions. Moreover, our LLM-powered technique resolves 7 regressions if not provided with error messages, and 9 regressions if provided with the error messages. As a comparison, the FDF algorithm from [3] can locate 6 root causes of regressions, while Top-2 algorithm can locate 8 causes.

In JavaScript experiments, our LLM-powered technique can successfully locate 9 causes of regressions when provided with error messages, which outperforms all other techniques. Our deepest-change algorithm and first-change algorithm resolve 6 and 8 regressions respectively, and also outperform existing techniques from [3]. As a comparison, the FDF algorithm resolves 6 regressions, while Top-2 algorithm only resolves 3 regressions. However, our LLM-powered technique performs differently with and without error messages. When not provided with error messages, LLM-powered technique only resolves 3 regressions.

We also present the overhead of our techniques in Table V. The transformation time indicates the time for modifying the source code to insert tracing statements. The runtime of modified program indicates the time for running the unit tests. The analysis time indicates the time for analyzing the differences between the base version and the faulty version. Specifically, all reported times correspond to the combined time of the base and faulty versions. The size of trace files is also the combined size of the base and faulty version. For easier understanding, we show the aggregation time of all steps in Figure 11. It can be seen that in most cases, techniques that trace source code consume less time than tracing inputs/outputs. This is because less time is spent on program modification and program execution. However, in some cases, analyzing source code change might take slightly longer time than analyzing changes between inputs and outputs. Moreover, tracing source code consumes less memory than tracing I/O. This is because we do not need to record the source code every time a function gets executed. Instead, we just need to record the identifier of the function. The source code of the executed functions only needs to be saved once.

There are five regressions where all techniques fail to resolve, including two cases in tornado by Python with id 2 and 12, one case in black by Python with id 19 and two cases in Express by JavaScript with id 13 and 27. We look into these cases and find that the failures of these cases are related to large-scale refactoring and many scattered changes in single commit.

For example, both bug 2 and bug 12 from tornado introduce over a thousand changes in lines of code. These changes make the execution traces vary a lot between the base version and the faulty version. It is difficult for both heuristic algorithms and an LLM to locate the correct cause of the regression from such a lot of changes. Although the large amount of changes does not necessarily mean the impossibility of locating the causes of regressions, it surely increases the difficulty for analyzing the tracing results and locating the true cause.

Specifically, as mentioned by [3], Bugfox cannot deal with cases in which random data such as IP addresses or port

[4]https://zenodo.org/records/15203310

TABLE II: Number of files and lines changed in each regression

(a) Number of files and lines changed in Python regressions

Bug id	No. files changed	No. lines changed
youtube-dl-19	3	15
youtube-dl-20	3	79
youtube-dl-25	4	34
youtube-dl-26	4	34
youtube-dl-29	3	15
youtube-dl-37	3	18
youtube-dl-41	5	23
tornado-2	29	2618
tornado-9	14	155
tornado-12	19	1487
black-3	2	28
black-19	3	113

(b) Number of files and lines changed in JavaScript regressions

Bug id	No. files changed	No. lines changed
Express-8	4	35
Express-9	6	418
Express-13	5	186
Express-16	4	232
Express-18	4	63
Express-27	5	196
ESLint-10	12	1612
ESLint-134	4	93
ESLint-307	4	260
hessian.js-2	8	112
hessian.js-8	7	156

TABLE III: The results for regressions in Python

Bug id	Tracing I/O (FDF)	Tracing I/O (Top 2)	Deepest code change	First code change	LLM only	LLM with error message
youtube-dl-19	no	yes	yes	yes	no	yes
youtube-dl-20	yes	yes	yes	yes	yes	yes
youtube-dl-25	no	yes	no	yes	yes	yes
youtube-dl-26	no	yes	no	yes	yes	yes
youtube-dl-29	yes	yes	yes	yes	yes	yes
youtube-dl-37	yes	yes	yes	yes	yes	yes
youtube-dl-41	yes	yes	yes	yes	yes	yes
tornado-2	no	no	no	no	no	no
tornado-9	yes	yes	yes	yes	yes	yes
tornado-12	no	no	no	no	no	no
black-3	yes	no	yes	yes	no	yes
black-19	no	no	no	no	no	no
total	6	8	7	9	7	9

numbers are generated. Such randomly-generated data will interfere the judgement of algorithms when tracing I/O. Our techniques are not influenced by these randomly-generated data. An example is the bug 9 from the project Express, which contains a lot of randomly-generated values. Our techniques successfully locate the cause of this regressions while algorithms tracing I/O fail.

5. Discussion

We empirically show that tracing source code change instead of tracing inputs and outputs of functions provides better accuracy in locating the cause of software regressions. Also, we empirically show that an LLM-powered technique with error messages achieves the best accuracy. In this section, we provide a discussion about our approach, as well as the threats to validity.

5.1 Tracing source code instead of I/O

There are several reasons about why we choose to compare source code changes instead of comparing I/O of functions. Generally, we think locating the cause of a software regression is about locating the key change in source code that leads to the malfunctioning of a feature. First, some functions do not have an explicit input or output. As an example, the function provided in Section 2 does not have an explicit return value. In such cases, the inputs or outputs never change and will cause false positives when analyzing the changes of I/O. Such cases often happen when a function modifies some global values or tries to pass wrong arguments to another function. When a function invokes another function but passes wrong arguments to that function, the I/O of the invoked function changes. The invoked function is reported as the cause of the regression by tracing I/O. However, this is wrong. The function that passes wrong arguments, instead of the function being invoked, should be blamed. In this case, tracing inputs/outputs would report a wrong cause of the regression, which is not what we expect. The algorithms mentioned in [3] do not consider such cases where inputs and outputs never change.

Second, tracing inputs and outputs may generate large tracing logs. Our statistics in Table V demonstrates the reduction in memory usage achieved by tracing source code. Some benefits are introduced by smaller tracing logs. One benefit is the time

TABLE IV: The results for regressions in JavaScript

Bug id	Bugfox [3] (FDF)	Bugfox [3] (Top 2)	Deepest code change	First code change	LLM only	LLM with error message
Express-8	yes	yes	yes	yes	no	yes
Express-9	no	no	no	yes	no	yes
Express-13	no	no	no	no	no	no
Express-16	no	no	no	yes	no	yes
Express-18	no	yes	no	yes	no	yes
Express-27	no	no	no	no	no	no
ESLint-10	yes	yes	yes	yes	yes	yes
ESLint-134	yes	no	yes	yes	no	yes
ESLint-307	yes	no	yes	no	no	yes
hessian.js-2	yes	no	yes	yes	yes	yes
hessian.js-8	yes	no	yes	yes	yes	yes
total	6	3	6	8	3	9

TABLE V: The overhead for regressions in JavaScript

Bug id	Transformation time		Runtime of modified program		Analysis time		Size of trace files	
	Tracing I/O	Tracing source code	Tracing I/O	Tracing source code	Bugfox	Our tool (LLM-powered)	Tracing I/O	Tracing source code
youtube-dl-19	49.7s	43.1s	1.6s	1.4s	1.25s	2.6s	50.9 MB	6.8 MB
youtube-dl-20	35.8s	32.3s	0.2s	0.2s	0.24s	2.3s	3 MB	0.34 MB
youtube-dl-25	43.4s	38.5s	0.7s	0.5s	0.31s	3.0s	5.4 MB	0.86 MB
youtube-dl-26	44.3s	39.1s	0.7s	0.4s	0.18s	3.1s	5.4 MB	0.84 MB
youtube-dl-29	36.1s	32.6s	1.0s	0.5s	0.28s	1.5s	4.6 MB	0.36 MB
youtube-dl-37	26.5s	24.2s	1.3s	0.6s	0.27s	1.6s	2.2 MB	0.04 MB
youtube-dl-41	26.7s	25.6s	0.6s	0.2s	0.13s	2.7s	1.9 MB	0.08 MB
tornado-2	25.8s	24.1s	26.1s	5.7s	6.07s	3.2s	253.8 MB	2.5 MB
tornado-9	36.6s	29.1s	0.8s	0.3s	0.93s	0.8s	33.4 MB	0.56 MB
tornado-12	22.1s	21.1s	2.1s	0.7s	1.09s	3.5s	51.8 MB	2 MB
black-3	8.2s	6.7s	3.4s	0.7s	0.91s	1.6s	53.4 MB	0.49 MB
black-19	4.1s	3.6s	19.2s	1.1s	10.73s	3.7s	670.1 MB	0.36 MB
Express-8	4.2 s	3.9s	16.2s	8.3s	1.46s	3.4s	125 MB	0.27 MB
Express-9	4.1 s	4.1s	3.1s	1.2s	0.59s	2.4s	41 MB	0.34 MB
Express-13	3.5 s	3.4s	17.1s	11.1s	1.85s	3.2s	93 MB	0.31 MB
Express-16	2.7 s	2.2s	20.5s	10.2s	2.86s	1.7s	132 MB	0.29 MB
Express-18	3.9 s	3.7s	12.6s	7.2s	1.44s	2.5s	80 MB	0.36 MB
Express-27	3.6 s	3.5s	21.4s	15.8s	3.26s	2.4s	99 MB	0.58 MB
ESLint-10	17.9 s	15.9s	1.1s	0.8s	0.43s	1.4s	24 MB	0.94 MB
ESLint-134	15.5 s	15.2s	2.3s	1.6s	0.56s	2.4s	7 MB	3.6 MB
ESLint-307	18.2 s	17.5s	1.1s	0.9s	0.55s	2.2s	8 MB	1.5 MB
hessian.js-2	1.2 s	1.1s	1.2s	0.9s	0.47s	1.9s	39 MB	0.08 MB
hessian.js-8	1.1 s	1.1s	1.5s	0.6s	0.54s	1.8s	41 MB	0.16 MB

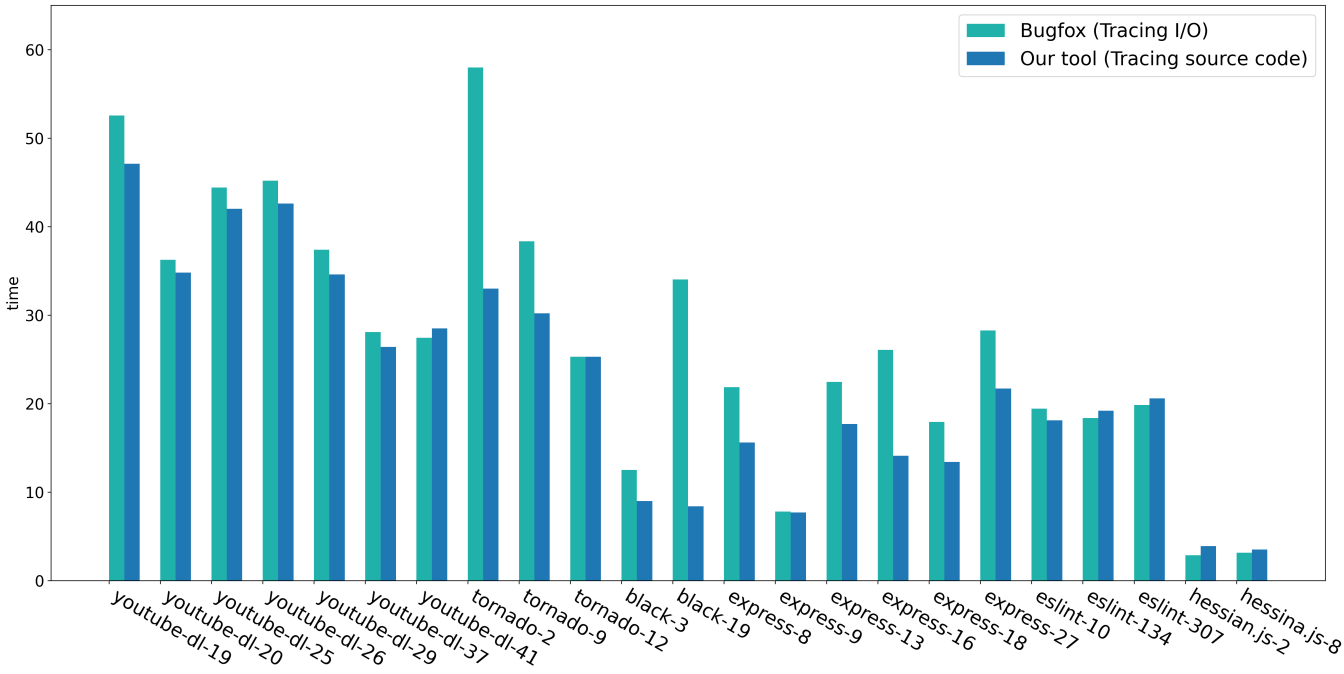


Figure 11: The aggregation time for analyzing a regression.

consumption. It can be seen that analyzing source code spends less time than analyzing I/O to locate the causes of regressions in most cases. Another benefit is that analyzing source code improves the accuracy. For example, the regression from Express with bug id 16 generates a large tracing log when tracing I/O, but a relatively smaller one when tracing source code. However, both algorithms analyzing I/O fail while our algorithm analyzing source code succeeds in locating the cause.

Third, as mentioned in [3], randomly-generated values sometimes prevent tracing I/O from successfully locating the true causes. Tracing source code is not affected by such randomly-generated values.

Moreover, tracing inputs and outputs is dependent on the serialization feature of each programming language. However, it may not be friendly to some off-the-shelf engines. For example, a generator function in Python is difficult to be serialized without changing its inner states, thus difficult to be traced. Without tracing such features, the accuracy may be compromised. Although we can develop some techniques to bypass these limits, it is out of the scope of this paper. In this paper, we intend to develop a lightweight and easy-to-use tool for detecting the causes of software regressions. Modifying compilers or interpreters would be a harder and more systematic job, which could be the future work of this paper.

Although we prefer source code changes instead of input/output changes, it is still necessary to trace program execution dynamically because we need the execution order of functions to locate the causes of regressions. In this task, the execution order helps us exclude some functions with source code changes but are irrelevant to the current test case, and helps us better locate a true cause of the regression. In other words, a dynamic execution order helps narrow the decision space.

5.2 The LLM-powered technique

A large language model combined with error messages shows better accuracy in locating the causes of regressions. From Table III and Table IV, it can be seen that most regressions can be resolved by our deepest-change-with-the-same-position and first-change-with-the-same-position algorithms. However, when these two algorithms do not align with each other, it is difficult to make a decision about the true cause. Large language models have demonstrated abilities in understanding source code [9], and dominated many code-related tasks [10], [11]. In this regard, we try to examine whether utilizing an LLM to help determine the true cause of software regressions is feasible.

Simply adopting an LLM is arbitrary and does not provide good accuracy. We provide error messages as supporting materials to an LLM, and find out that it achieves better accuracy than heuristic algorithms. The experiment shows that naively using an LLM is not enough for locating the true causes of regressions. Error messages help an LLM better analyze the source code and make the decisions.

The window size of the sub-execution trace is another concern. From our experiements, we find that a smaller window size would decrease the accuracy when the nested function call is long. However, a larger window size does not necessarily increase the accuracy. When the window size is as large as 10 before the starting point and 50 after the starting point, the accuracy decreases again due to LLM hallucination. Our

choice of window size is based on our experiment results. A systematic analysis of the window size and prompting engineering might be the future work.

5.3 Threats to validity

There are some threats to validity of our work. First of all, the number of regressions in our dataset is small. Although we cover two programming languages and collect these regressions from real-world projects, it is still limited to evaluate tools specifically for locating the causes of regressions. This is due to the difficulty of identifying software regressions in publicly available datasets. Further benchmarking with more real-world regressions as well as more programming languages is needed to conduct a more comprehensive evaluation.

Second, there are still some regressions that our techniques cannot resolve. One reason is the large-scale refactoring. When there are massive scattered changes and the execution traces vary a lot, it is difficult for both heurictic algorithms and LLMs to locate the true causes. Also, semantically-equivalent changes may cause false positives by our algorithms. Although we empirically show that our techniques perform better than previous techniques, the boundary conditions of our techniques associated with these refactorings require further research.

6. Related Works

Regression testing techniques have getting more important because of the emergence of new approaches for developing software, such as the agile movement [1]. Regression testing consists of rerunning the previously executed tests when the software under test evolves to verify that no new failures, or regressions, have been introduces [2]. Unit testing frameworks [6], [12], [13] play an important role in regression testing.

A number of approaches have been investigated to make regression testing more effective and efficient [2]. For example, test case prioritization [14] determines an execution ordering that ideally gives precedence to the most effective test cases; test case selection [15] takes a sample of test cases for execution generally based on the recently introduced changes; and test suite reduction [16] aims at removing reduntant test cases according to some criterion, for instance code or requirements coverage. Moreover, [17] provides techniques for improving regression testing in continuous integration development environments.

Although detecting regressions is well studied [18], localizing the causes of regressions requires further exploration. A number of techniques focus on preventing regressions before release [19], fewer tools effectively address the causes of regressions post-release [20]. [21] provides a tool for regressions in C/C++ software. This work traces predicates and reports suspicious differences between the base version and the upgraded version by using GDB debugger. The drawback of this work is that it is unable to implement universal tracing with complete data serialization because of the limitations of a low-level and procedural programming language. As a result, only a small subset of programs can be applied using this technique to localize the causes of regressions. [22] provides a tool that adopts machine learning techniques to debug regressions. It utilizes historical data from a version control system and the results from functional debugging. They train models to rank revisions based on their probability to be the cause of a certain failure. They use an undisclosed dataset to test their tool. One recent work is done by [3], which traces dynamic information during program execution, including changes of arguments and return values. They implement their idea in JavaScript and can successfully address the causes of a few regressions. However, the accuracy is inferior to our proposal.

7. Conclusion

In this paper, we provide techniques to improve the accuracy of locating the root cause of software regressions compared with the existing approach. We achieve this by tracing the order of function execution dynamically, and comparing the source code of functions in the execution trace. In detail, we provide three techniques. Two of them are based on heuristic algorithms, namely the first code change with the same position and the deepest code change with the same position. The other one is powered by a large language model combined with error messages. Specifically, our LLM-powered technique achieves the best accuracy compared with other techniques on both JavaScript dataset and Python dataset. We also extend the existing dataset from JavaScript to Python for better comparison, and conduct the experiments on both languages.

Acknowledgment

The work is supported by JSPS KAKENHI ...

References

[1] R. H. Rosero, O. S. Gómez, and G. Rodríguez, “15 years of software regression testing techniques — a survey,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 05, pp. 675–689, 2016. [Online]. Available: https://doi.org/10.1142/S0218194016300013

[2] R. Greca, B. Miranda, and A. Bertolino, “State of practical applicability of regression testing research: A live systematic literature review,” *ACM Comput. Surv.*, vol. 55, no. 13s, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3579851

[3] Y. Hu, H. Ishibe, F. Dai, T. Yamazaki, and S. Chiba, “Bugfox: A trace-based analyzer for localizing the cause of software regression in javascript,” in *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 224–233. [Online]. Available: https://doi.org/10.1145/3687997.3695648

[4] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1556–1560. [Online]. Available: https://doi.org/10.1145/3368089.3417943

[5] T. Holowaychuk, StrongLoop *et al.*, "Express - fast, unopinionated, minimalist web framework for node.js," https://expressjs.com/, 2024.

[6] O. Foundation, "Mocha - the fun, simple, flexible test framework," https://mochajs.org/, 2024.

[7] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of nlp and software engineering: A survey on language models for code," 2024. [Online]. Available: https://arxiv.org/abs/2311.07989

[8] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark of javascript bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 90–101.

[9] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639187

[10] A. Ni, P. Yin, Y. Zhao, M. Riddell, T. Feng, R. Shen, S. Yin, Y. Liu, S. Yavuz, C. Xiong, S. Joty, Y. Zhou, D. Radev, and A. Cohan, "L2ceval: Evaluating language-to-code generation capabilities of large language models," 2023.

[11] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang, X. Che, Z. Liu, and M. Sun, "Repoagent: An llm-powered open-source framework for repository-level code documentation generation," 2024.

[12] pytest-dev Team, "pytest: helps you write better programs," https://pytest.org, 2024.

[13] K. Beck, E. Gamma, D. Saff, and K. Vasudevan, "JUnit - a programmer-friendly testing framework for java and the jvm," https://junit.org/junit5/, 2024.

[14] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing," *Inf. Softw. Technol.*, vol. 93, no. C, p. 74–93, Jan. 2018. [Online]. Available: https://doi.org/10.1016/j.infsof.2017.08.014

[15] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Comput. Surv.*, vol. 50, no. 2, May 2017. [Online]. Available: https://doi.org/10.1145/3057269

[16] S. U. Rehman Khan, S. P. Lee, N. Javaid, and W. Abdul, "A systematic review on test suite reduction: Approaches, experiment' s quality evaluation, and guidelines," *IEEE Access*, vol. 6, pp. 11 816–11 841, 2018.

[17] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 235–245. [Online]. Available: https://doi.org/10.1145/2635868.2635910

[18] H. Srikanth and M. B. Cohen, "Regression testing in software as a service: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 372–381.

[19] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, p. 184–208, apr 2001. [Online]. Available: https://doi.org/10.1145/367008.367020

[20] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 199–209. [Online]. Available: https://doi.org/10.1145/2001420.2001445

[21] F. Pastore, L. Mariani, and A. Goffi, "Radar: A tool for debugging regression problems in c/c++ software," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1335–1338.

[22] D. Maksimovic, A. Veneris, and Z. Poulos, "Clustering-based revision debug in regression verification," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015, pp. 32–37.