FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Integration of Fault Localization into your GitHub Repository

Hugo Filipe Ribeiro Paiva de Almeida

Master in Software Engineering

Supervisor: Prof. José Carlos Medeiros de Campos

Second Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu

October 09, 2023

Integration of Fault Localization into your GitHub Repository

Hugo Filipe Ribeiro Paiva de Almeida

Master in Software Engineering

Approved in oral examination by the committee:

President: Prof. Nuno Honório Rodrigues Flores

Referee: Prof. Alcides Miguel Cachulo Aguiar Fonseca

Referee: Prof. José Carlos Medeiros de Campos

October 09, 2023

Resumo

Com a crescente complexidade e escala do software, existe uma forte necessidade de técnicas que auxiliem os desenvolvedores de software a localizar falhas com o mínimo de intervenção humana possível.

O objetivo desta dissertação é analisar o uso de abordagens de localização de falhas baseadas em espectro para ajudar a descobrir falhas em programas Java, bem como o uso de *bots* no ciclo de vida do desenvolvimento de um software. As técnicas de localização de falhas baseadas em espectro foram escolhidas na área de pesquisa de localização de falhas de software devido aos seus baixos custos de execução e popularidade. Três ferramentas (GZoltar, FLACOCO e Jaguar) destacaram-se como as principais escolhas para a localização de falhas baseada em espectro em Java, de acordo com a pesquisa, e embora todas produzissem resultados comparáveis, o GZoltar foi preferido.

Foi criada uma *Action* do GitHub que, quando integrada com o GZoltar, permite a análise de relatórios de localização de falhas baseada em espectro em qualquer repositório Java no GitHub. O resultado é um relatório detalhado das linhas de código potencialmente com falhas, personalizável pelo utilizador.

Esta *Action* foi avaliada tanto em um repositório de exemplo como em vários projetos *open-source*. Embora a integração tenha sido bem sucedida no repositório de exemplo, as limitações do GZoltar impedem a sua integração na maioria dos projetos *open-source*, destacando a necessidade de atualizações e testes adicionais de compatibilidade.

Abstract

With the increased complexity and scale of software, there is a strong demand for techniques to guide software engineers to locate faults with less human intervention as possible.

The purpose of this dissertation is to look into the usage of Spectrum-based Fault Localization approaches to help discover faults in Java programs, as well as the use of bots in the software development lifecycle. Spectrum-based Fault Localization techniques were found to be chosen in the research area of software fault localization due to their low execution costs and popularity. Three tools (GZoltar, FLACOCO, and Jaguar) stood out as the top choices for spectrum-based fault localization in Java according to the research, and even though all produced comparable outcomes, GZoltar was preferred.

A GitHub Action was created that, when integrated with GZoltar, allows analysis of Spectrum-based Fault Localization reports in any Java repository on GitHub. The outcome of it is a detailed report of potentially faulty lines of code, customizable by the user.

This action is tested in both a sample repository and several open-source projects. While successful integration is achieved with the sample repository, limitations of GZoltar hinder its integration with most open-source projects, highlighting the need for updates and further compatibility testing.

Acknowledgements

I would like to thank my family for all the support, encouragement, and financial investment they have provided me throughout these past years of my education.

I thank my colleagues and friends who have helped me both inside and outside the academic environment, making the completion of this stage easier.

I thank Carolina for always being present in every moment of my academic life, supporting me and encouraging me to never give up.

Lastly, a big thank you to my advisors, Prof. José Carlos Medeiros de Campos and Prof. Rui Filipe Lima Maranhão de Abreu, for their availability, trust, and motivation during this final stage of my Master's degree.

Hugo Filipe Ribeiro Paiva de Almeida

"What would you attempt to do if you knew you could not fail?"

Robert H. Schuller

Contents

List of Figures

List of Tables

Listings

Abbreviations and Symbols

GDP	Gross domestic product
SBFL	Spectrum-Based Fault Localization
CI	Continuous Integration
ESHS	Executable Statement Hit Spectrum
HTML	HyperText Markup Language
CLI	Command-Line Interface
API	Application Programming Interface
IDE	Integrated Development Environment
KLOC	Thousands of Lines Of Code
AST	Abstract Syntax Tree
JSON	JavaScript Object Notation
CSV	Comma Separated Values
duas	Definition-Use Associations
DRT	Debugging strategy based on Requirements of Testing
PAT	Personal Access Token
SHA	Secure Hash Algorithm

Chapter 1

Introduction

1.1 Context

Software faults in software systems have significant complications, most notably huge financial losses. In 2002, software errors were estimated to cost the United States economy $59.5 billion annually (0.6% of the Gross domestic product (GDP)) [1]. Since then, the cost has grown where over 50% of the cost of fixing these bugs is passed to the users, while the remaining is imputed to the developers and vendors.

With the increased complexity and scale of software, there is a strong demand for techniques to guide software engineers to locate faults with less human intervention as possible.

Researchers have been proposing automatic techniques to help with the Software Fault Localization process, i.e., identify the location of faults in a program. Several techniques have been developed, such as Spectrum-based Fault Localization (SBFL) [1], which uses the program spectrum. It details the execution information of a program from certain perspectives, on both faulting and non-faulting program executions of test cases, to track program behavior and narrow the search.

Some tools aimed at SBFL in the Java programming language have emerged, like, GZoltar [12], FLACOCO [2], and Jaguar [3]. However, there is an absence of successful transitions of this technology in the software industry [8].

1.2 Problem

Nowadays, many developers still debug programs manually [13], which is a cumbersome, costly, and time-intensive task. The continued use of traditional fault localization techniques contributes to the high cost of software failures. Breakpoints, logs, and assertions are outdated in a hugely developed world that offers highly autonomous tools for those tasks.

Using the breakpoints as an example, a developer still needs to select several specific points across the code to make the program stop and analyze the data produced before stopping. In the

worst-case scenario, faults may not be immediately identified at the first breakpoint, increasing the time spent on this type of analysis.

Nonetheless, even with the advent of automatic techniques for SBFL, the lack of use in the software industry is strongly notorious, primarily due to its deficit of integration with the systems.

Accordingly, there is a need to create a simple and quick-to-setup tool capable of applying the automated techniques mentioned above in the continuous integrations environment, as has been suggested previously by practitioners [14]. In addition, combining this tool with a code hosting service becomes an asset in automating the debugging process. This reduces the time spent on fault detection, optimizing the entire debugging process and centralizing all code-related information on a single platform.

1.3 GZoltar Automatic Debugging for GitHub Actions

GZoltar Automatic Debugging for GitHub Actions is a tool that aims to provide an automated way of applying SBFL techniques in the Continuous Integration (CI) environment. It is a GitHub Action that can be easily integrated into any GitHub repository, allowing any developer to run the tool on every push to a repository. The outcome is a detailed report of the suspicious lines of code detected, according to the settings set by the user.

1.4 Contributions

This dissertation brought 3 significant contributions to the software development ecosystem:

1. One of the primary contributions of this work is the development of the GZoltar Automatic Debugging Action. This action is part of a novel approach that automates the feedback process in fault localization. By leveraging the power of Spectrum-Based Fault Localization techniques, GZoltar Automatic Debugging uses GZoltar results and provides automated feedback on potentially faulty code locations. This automation saves developers considerable time and effort in debugging, making it an invaluable tool for software development teams.

2. Another significant contribution of this work is the integration of the GZoltar Automatic Debugging Action into open-source projects. The integration process involved configuring CI pipelines and adapting the tool to work with open-source projects. This integration enabled the tool to be applied to real-world software projects, providing valuable insights into the effectiveness of the automated feedback approach.

3. While the integration of GZoltar Automatic Debugging into open-source projects was partly possible, it was not without challenges. The process involved overcoming various hurdles related to coding style and complexity differences across different projects. Furthermore, some challenges were impossible to overcome, making this discussion a valuable insight into the future development of similar tools.

1.5 Structure

Chapter 1 presents a brief introduction and the motivation of the problem as well as the objectives defined and contributions.

Chapter 2 provides the background information needed to understand the potential problem solutions.

Chapter 3 addresses the existing solutions to culminate the problems raised.

Chapter 4 describes the architectural choices, the specification and limitations of the developed solution as well as a usage example.

Chapter 5 presents a qualitative evaluation of the developed solution with a similar tool and a detailed discussion of the results obtained in a real-world scenario implementation.

Chapter 6 discusses the objectives, their satisfaction and conclusions reached. Interesting suggestions for future work are also presented.

Chapter 2

Background

2.1 Software Fault Localization

Fault Localization, Fault Understanding, and Fault Repair are the phases involved in software debugging. Software Fault Localization consists in identifying the locations of faults across a program. This technique is considered one of the most tedious, time-consuming, expensive, and critical activities during program debugging.

The increase in complexity and scale of software not only caused a strong demand for techniques able to orient the software engineers to locate faults with less human intervention but also emerged the investigation around Software Fault Localization techniques as well as paper publications. Fig. 2.1 reports the number of publications by year between 1977 and 2014, which has been growing significantly.

Software Fault Localization Techniques can be grouped into two main groups: Traditional Techniques and Advanced Techniques.

2.1.1 Traditional techniques

Traditionally, developers have found flaws in software by resorting to old and standard techniques such as Program Logging, Assertions, Breakpoints, and Profiling [1].

Program logging compacts statements to produce logs. Any kind of *print* is used by developers to examine the program's condition to diagnose the cause of failure.

Assertions are constraints added to a program's logic that must be true for the software to work correctly. The program stops when they are falsely evaluated, signaling developers a problem.

Breakpoints are used to examine the current state of a program. Its operation is based on pausing the execution when a condition is met, allowing the state gathered at that stage to be analyzed or changed and allowing the developer to continue the execution. When triggered based on variables value, they are classified as Data Breakpoints. If triggered by a predicate specified by the user, they are classified as Conditional Breakpoints.

Profiling is used to detect unexpected execution frequencies of functions, identify memory leaks, and others by analyzing runtime metrics such as execution speed, memory usage, etc.

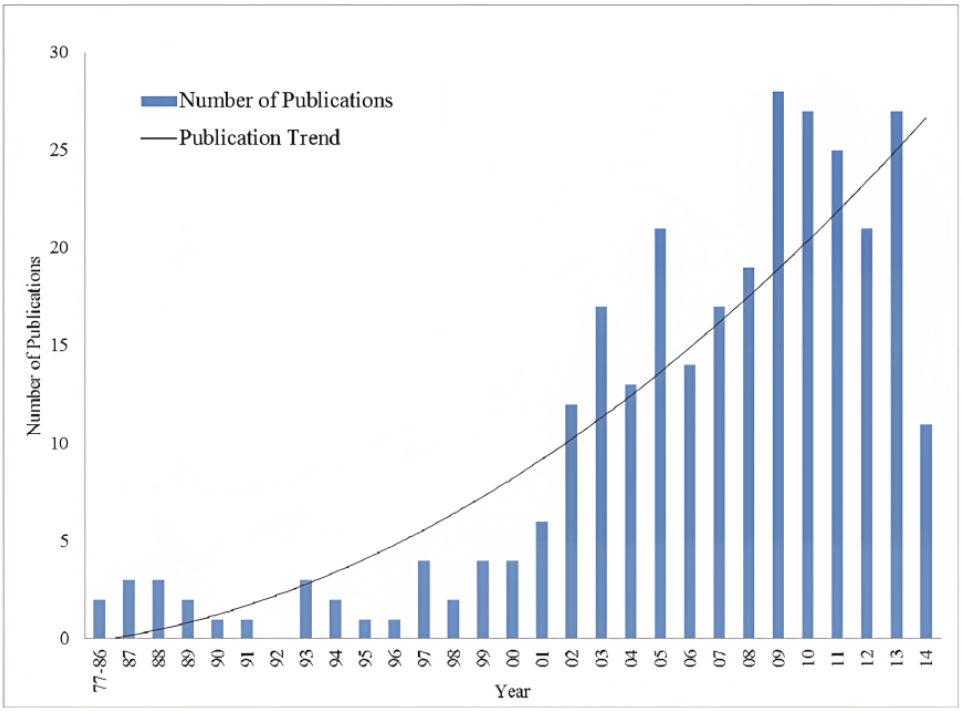


Figure 2.1: Publications on Software Fault Localization from 1977 to November 2014 [1]

Although these techniques have been helpful in the past, the massive size and scale of software systems today have made them less effective in isolating the root causes of failures.

2.1.2 Advanced techniques

Many advanced techniques have emerged based on the idea of causality, which characterizes the relationship between events/causes (bugs) and phenomenon/effect (execution failures). Based on the survey provided by Wong et al. [1], Advanced Software Fault Localization techniques can be classified into eight categories: slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based, and other miscellaneous techniques.

Slice-based techniques are grounded on the idea of reducing the search domain while programmers locate faults in programs. Static Slicing is one of the examples, and it focuses on finding the static slice of the code associated with a variable in case a test case fails due to an incorrect variable value in a statement. Derived from this idea, Dynamic Slicing was also introduced to avoid the generation of slices containing statements that could affect that variable but are not executed, creating misleading data. Other Slice-based techniques have been introduced, from Relevant Slicing, which deals with a limitation of the previous technique regarding the inability to capture execution omission errors, to Execution Slicing, which is based on data-flow tests each one of them created in an attempt to improve the existing techniques.

Spectrum-based techniques resorts to the program spectrum to track a program's behavior. This spectrum details the execution information of a program, such as execution information for conditional branches. By looking at the Code Coverage, also known as Executable Statement

Hit Spectrum (ESHS), it is possible to gather the program parts covered during testing. If a test fails, this information can be handy in identifying the components involved in the failure, allowing these techniques to narrow the search for a fault. Most spectrum-based techniques assign a suspiciousness score to each statement based on the program spectrum.

Program state-based techniques use variables and their values during program execution to locate faults. Each technique avails the program state in distinct manners:

- Relative Debugging compares a runtime state of a program with another "reference" version of the same program
- Delta Debugging contrasts the state of the program between successful and failed test executions by replacing variables values from the successful runs in the failed ones
- Predicate Switching changes the program state to force an alteration in branches executions on failed tests

In short, most of them manipulate or gather information from the program state to reduce the fault's search radius.

Machine learning-Based techniques are adaptive by improving based on the data on which models are created. Implementations using back-propagation neural networks, radio basis function networks, and others can be found in different research. In the vast majority, the coverage data of each test case is used to train the networks, and then a statement is used as input to the trained network to obtain the likelihood of containing a fault.

Data mining-based techniques also produce a model based on data extracted from the program, which can lead to finding hidden patterns due to the massive volume of information that can be used on the model. From program state to execution traces, different information can be applied in statistical analysis, understanding of possible association rules, etc.

At last, Model-based techniques, by assuming that a correct model of each program being analyzed is available, look for differences between the model expectation and the observed behavior of the program under analysis. If the model being used was not generated from a program with bugs, it is expected that these differences might lead to finding faults. By using this methodology, different models can be derived from program information. Dependency-based, abstraction-based, and value-based models are examples of extracting different information to create a model. Respectively, each uses dependency between statements, abstract interpretation of components, and data-flow information to create the models.

Spectrum-Based Fault Localization techniques have been shown to be promising due to their low execution costs [15]. Its popularity does not show the opposite, either. SBFL has been used in 41% of research publications in the field of Software Fault Localization [16], becoming an easy choice over other existing techniques.

2.2 Spectrum-Based Fault Localization (SBFL)

The first SBFL studies focused only on failed test cases that were proven ineffective [1]. Nowadays, the analysis is based on both failing and passing tests to emphasize the contrast between them.

Techniques such as Set Union and Set Intersection were experienced [17] focusing only on the source code executed by failed test cases and excluding the code executed by all successful test cases, respectively. Nearest Neighbor [17] evolves from that by looking at the differences between a failed test and a successful test that are similar given a distance function based on the generated coverage matrix. If a fault is in the difference set, it is located. If not, the process continues using a program dependence graph.

Similarity Coefficient-based techniques also emerged with the logic that the closer the execution pattern of a statement is to the failure pattern of all test cases, the more likely that statement is faulty. These techniques have different formulas that are used to quantify this proximity. This quantification can be interpreted as the suspiciousness of a failure in a statement.

Ochiai [1] is included in the Similarity Coefficient-based techniques by utilizing the coverage and execution results to compute the suspiciousness of each statement. The Equation 2.1 used by Ochiai is composed of the number of failed test cases a statement covered (N_{CF}), the number of successful test cases a statement covered (N_{CS}) and the number of failed (N_F) test cases.

$$Suspiciousness(Ochiai) : \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}} \tag{2.1}$$

A wide range of other Similarity Coefficient-based techniques have been proposed. Names like O, O^P, Tarantula, Ochiai2 (an extension of Ochiai), Dennis, and at least 30 more others are existing techniques [1] whose formula for calculating suspiciousness changes, but the underlying operation is the same.

The application examples of using the program spectrum are extensive and do not stop there. Program Invariants Hit Spectrum-based utilizes the coverage of program invariants, i.e., logical assertions that must hold true during the execution of a program, to locate bugs in possible violations of them. Predicate Count Spectrum-based keeps track of the execution of predicates, the operators that output boolean values, to identify program behaviors that might contain bugs. Method Calls Sequence Hit Spectrum-based, on the other hand, uses the sequence of method calls covered during the test executions. Time Spectrum-based records the methods execution time and creates a behavior model from the time spectra collected with the successful test case executions, considering disparities from these models in failed test case executions as suspicious.

These techniques are not the only ones to apply SBFL, nor will they be the last. SBFL's popularity within the scientific community will undoubtedly continue to bring more news.

Although there are several options, the Ochiai technique has been shown to be one of the leading choices. It has not only obtained better results than the Tarantula, Jaccard, and Ample

Table 2.1: Java example of calculating suspiciousness using the Ochiai technique

	Triangle type function code with a fault at *l2*	T1	T2	T3	T4	Suspiciousness
l1	public static String triangleType(int a, int b, int c) {	x	x	x	x	0,5
l2	if (a == b || b != c || a == c) {	x	x	x	x	0,5
l3	return "Isosceles";	x				1,0
l4	} else if (a == b && b == c) {		x	x	x	0,0
l5	return "Equilateral";		x			0,0
l6	} else {			x	x	0,0
l7	return "Scalene";			x	x	0,0
	...					
	Test Execution Result	Pass	Pass	Fail	Pass	

techniques in a study by Abreu et al. [18], but is also considered as one of the top 3 among 42 techniques evaluated concerning the technique's accuracy and user effort required when analyzing the diagnosis generated by applying SBFL [19].

As mentioned previously, applying the Ochiai technique to compute the suspiciousness of each statement is relatively easy. When using Equation 2.1, it is only necessary to identify the number of successful and failed test cases in each line of code covered and the total number of failed tests.

Table 2.1 applies this calculation in every statement of a function where its original purpose was to identify the type of a triangle given the size of each of its sides. Bearing in mind that a triangle is isosceles when it has two equal sides, it is easy to identify that *l2* contains a fault. Furthermore, the code was tested with 4 test cases where the coverage of each statement could be observed by the marking with an x in each test. Finally, and to make the calculation possible, there is also an indication that the *T3* test was the only test that failed in the execution.

By looking purely at the suspiciousness value, a developer would soon be able to remove from his search any statement not in the first three lines, immediately reducing his effort. It should also be noted that although the statement with the most suspiciousness is not really where the fault is, taking into account the formula applied by Ochiai, it is very close. Also, it is clear why it has this highest value since no test successfully executed is covered by it.

2.3 Bots in Software Engineering

Bots, short for *Software Robots*, are software programs that can be as simple as automated scripts up to autonomous agents that execute tasks based on conditions [20] that would otherwise have to be done by humans. According to an Erlenhov et al. [21] study, there are two reasons to use them: productivity or quality of the work improvement.

Bots that aim to help developers in the software development process, also known as *DevBots*, can be programmed to do tasks from handling repository issues to the continuous integration or deployment pipelines. Accelerating code deployment, creating code documentation that lacks it, performing automated tests, and possibly making fixes on code based on failed tests are different ways they are created to aid.

Sankie [22], an Artificial Intelligence platform for DevOps, is a complex example of a bot that aids developers. By looking at data from repositories, such as changes reviewed by engineers, exception counts from post-deployment data, and others, Sankie is able to train itself and generate

recommendations. An example of the help generated is recommending a reviewer based on the author and set of files being changed.

This revolution can be seen extensively in collaborative code platforms. GitHub[1], for example, with the creation of products such as Github Actions [4], allowed not only the publication and execution of bots that help in the software development process but also the emergence of a marketplace of sharing and sale of services that make use of these.

Although these tools help improve humans' efficiency, they are not perfect. In most cases, there will always be limited flexibility since only tasks that have been programmed will be executed, and there is no capacity for creative thinking or problem-solving. Either way, the examples of help provided are extensive.

Wessel et al. [23], in a study on code review bots, discovered that the use of these bots increases the number of monthly merged pull requests, decreases the number of pull requests to be reviewed, and makes reviewing by developers more efficient. In addition, high code coverage, as well as quality, was enforced.

Kinsman et al. [24] researched the effects of automated workflows on repositories using GitHub Actions. Contrary to Wessel et al. [23], it has been noticed that the number of rejected pull requests has increased. However, according to Santhanam et al. [20], this is due to differences between most bots available through GitHub Actions and more specific bots for code review. Code review bots provide constructive feedback regarding the changes that need to be made for the pull request to be accepted. Most of the tools available through GitHub Actions only warn of something wrong without much feedback, leading to pull requests being rejected easily but with no corrections in sight.

2.3.1 Continuous Integration Tools

Continuous integration tools have grown in popularity among software engineers in recent years. According to a recent report by MarketsandMarkets [25], the continuous integration tools market size was $402.8 million in 2017 and is expected to reach $1139.3 million by 2023. This is due to the increased popularity of agile development methodologies, which stress iterative development and rapid releases.

Several continuous integration tools are available, each with its strengths and weaknesses. Some of the most popular ones are:

- **Jenkins** [26] - a widely-used open-source automation server for software building, testing, and deployment. Its extensive plugin system enables users to seamlessly extend and integrate functionality with other tools. Its highly configurable nature allows for customization on various operating systems and cloud platforms.

- **Travis CI** [27] - a cloud-based tool utilized for continuous integration, is highly preferred within open-source projects associated with GitHub. The configuration file format is facile

[1] https://github.com/

and delivers an intuitive approach to its user. Numerous platforms are supported alongside integrations with popular supplemental tools such as Slack to further enhance productivity and workflow efficiency.

- **CircleCI** [28] - is a cloud-based tool for continuous integration known for its fast and reliable building processes. It has a user-friendly graphic UI and offers support for numerous operating systems. In addition, it provides different integration with other tools, such as Kubernetes and Docker, to ensure simple construction and deployment of code.

- **GitLab CI/CD** [29] - an integrated CI/CD solution on the GitLab web-based Git repository manager that simplifies the whole software development process. With a straightforward and user-friendly configuration system, GitLab CI/CD provides cross-platform. It is also complemented with several other repository manager features, such as code review and issue tracking, to conveniently oversee all life cycle stages. As a result, users can enjoy the convenience of managing their entire workflow through one consolidated platform.

- **GitHub Actions** [4] - a CI/CD tool that enables users to automate their workflow, including compilations, tests, and direct deployments of code originating in one's GitHub repository. This feature offers a concise yet intuitive YAML-authored architecture for outlining tasks pertinent to the workflow. Additionally, there is support for various operating systems. Moreover, pre-set actions are available to expedite common processes such as code building and testing, and custom actions can also be created. In addition, its tight integration with GitHub provides not only the viability of controlling as well as configuring activities via the website but also access to numerous additional repository managing functions - analogous to GitLab CI/CD.

Although not covered in this listing, GitHub has another tool developers can use in CI environments, the GitHub Apps. It turns out that, unlike GitHub Actions, these require a server where they are configured, substantially increasing the difficulty of configuration concerning their counterpart, which is why it was not listed, and not considered in the development of this work.

2.4 Summary

This chapter presented the background information necessary to understand the work developed. It started by introducing Software Fault Localization, its importance, and the different techniques that exist. Then, it focused on Spectrum-Based Fault Localization, one of the most popular techniques, and the Ochiai technique, one of the most used. Finally, it presented the concept of bots, their use in software engineering, and the most popular continuous integration tools. The next chapter will detail a literature review of the existing Java SBFL tools as well as the only example of a bot for CI environments that uses them.

Chapter 3

Literature Review

The present chapter introduces the approaches of Spectrum-Based Fault Localization and Bots in Software Fault Localization, as well as their impact on the software development process. A literature review on Spectrum-Based Fault Localization tools for Java is also conducted, describing the most relevant tools and their characteristics.

3.1 Spectrum-Based Fault Localization (SBFL) Java Tools

Some SBFL tools have emerged to help developers find bugs easier and faster. Since Java is in the top-2 of primary programming languages according to a survey of more than 30,000 developers [30], it is natural that certain tools have also become available in its ecosystem. However, there is a barrier, mainly in their adoption.

Archana and Ashutosh Agarwal evaluated several SBFL tools [8], reiterating the great interest of the academic community in the SBFL topic, which has not migrated to the industry. Hence, the need to diversify how these tools are used.

Notwithstanding, these tools have proven their effectiveness and utility for developers and are just waiting to be used.

3.1.1 GZoltar

GZoltar is a Java toolset for fault localization [12]. It has the infrastructure to automatically instrument the source code of software programs to produce runtime data and analyze it to return a ranked list of faults candidates based on SBFL formulas. It is available as a command-line tool, a Visual Studio Code extension [31], and an Apache Ant [32] and Maven [33] plugin.

The main steps of its execution are:

1. Detection of test cases
2. Code instrumentation
3. Report generation

GZoltar starts with detecting all the test cases of the desired project. With the classes located, all the code in the project is instrumented to allow the code coverage process of the test classes by registering what statements are used by an execution. In this process, Javassist [34], a Java bytecode toolkit, is used to instrument the project and obtain the coverage traces when executing the unit tests with JUnit. This process can happen online or offline. In the online mode, instrumentation occurs as the tests are loaded into the Java Virtual Machine and executed. In the offline mode, there is a preprocessing step where all compiled classes are instrumented, generating new bytecode files with probes that will indicate the coverage of each line of code as the tests are executed.

After the test cases execution and instrumentation, a binary coverage matrix N x M is created, where N is the execution of a test case, and M is the different lines of code of the program. This matrix is capable of representing the coverage of each line of code by each test case as well as the result of the test execution.

With the collection of the coverage into the coverage matrix, GZoltar executes the SBFL algorithm with one or more desired formulas. GZoltar supports 16 SBFL formulas but also allows extensibility with custom implementations. It can produce both a textual and graphical representation (using HyperText Markup Language (HTML)) of the results.

The execution of GZoltar must be separated into these three steps because it avoids re-executing all the steps in case only information from some is required.

By being one of the oldest SBFL tool in Java still actively developed, GZoltar has been included in some experiments to prove its usefulness. Martinez et al. [35] while testing Astor, an automatic program repair library that uses Fault Localization techniques to focus on the statements of higher suspiciousness of bugs, tested it using GZoltar and was able to repair 33 bugs out of 224 automatically.

Gouveia et al. [36] performed an user study with 40 students of the Master in Informatics and Computing Engineering in the Faculty of Engineering of the University of Porto where they did a debugging task with and without the use of GZoltar in a 17 Thousands of Lines Of Code (KLOC) software, XStream. Most users of that study found GZoltar an intuitive and helpful toolset.

Xuan et al. [37] tested the Nopol repair system on 22 real-world Java bugs of Apache Commons Math[1] and Apache Commons Lang[2] with an average of 25 KLOC executable lines of code for each fault. One of the bugs analyzed needed as much as 75 KLOC to be executed. Nopol is an approach to the automatic repair of faulty condition statements, taking a test suite as input and generating a patch with a conditional expression as output to fix the fault. It uses Fault Localization techniques to rank statements according to their suspiciousness of containing bugs and check each candidate with suspiciousness over zero for a faulty condition statement. With the use of GZoltar, 17 out of 22 bugs were found and patched.

[1] https://commons.apache.org/proper/commons-math/
[2] https://commons.apache.org/proper/commons-lang/

3.1.2 FLACOCO

FLACOCO is a Fault Localization tool for Java that uses SBFL by applying code coverage to predict suspicious lines of code [2]. Built on top of JaCoCo [38], one of the most used and most reliable coverage libraries for Java, FLACOCO is made available through a Command-Line Interface (CLI) and a Java Application Programming Interface (API), supporting all Java versions.

Implementations and tools for SBFL have hardships in reaching the industry. With this in mind, FLACOCO aims to provide a simple interface, being effective regarding Fault Localization results, offering good performance, and being reliable over a wide range of Java and Java Virtual Machine versions.

The architecture of FLACOCO, described in Fig. 3.1, allows to understand how the tool works. Firstly, the user must run the tool through the CLI or the Java API. In both cases, a list of suspicious locations is the primary outcome. Then, there is a test detection process in which FLACOCO detects the tests to be executed and analyzed by scanning the compiled classes from the project under analysis. This process supports JUnit 3, 4, and 5 [39] and uses a new Java thread to discover tests.

With the tests detected, a separate process is used to execute the tests and instrument them to record the lines covered by each test case. The instrumentation of executed classes is done at the class loading time of a test execution, being fully provided by JaCoCo. Since instrumenting for coverage computation is a complex problem, where the tools responsible for that need to be updated whenever the Java Bytecode evolves, using JaCoCo, FLACOCO can support new Java versions by simply updating the JaCoCo version used. Furthermore, JaCoCo is a library with over a decade of existence. It is highly maintained and estimated to be used in at least 243,000 open-source projects in their software quality assurance processes [40], having enormous success. The execution of tests in an isolated process also allows FLACOCO to control the execution time of the tests (timing out if needed), avoid conflicts between classpaths from the app under testing and FLACOCO and, like previously stated, enable the instrumentation at class loading time. Although exception bugs are not an issue while executing tests, since JaCoCo inserts probes at control-flow statements of Java methods but not at the beginning of each line, executions might lead to misinformation. This happens because a line is marked as covered when a probe is placed after it is marked as executed. When an exception is thrown between two probes, some covered lines can be missing. To solve this, FLACOCO parses the stack trace of each exception thrown in order to collect the lines which have been executed.

Before computing the suspiciousness, FLACOCO collects and stores program spectra like test results (pass or fail), if the test throws an exception, and coverage per line for each executed class by the test. Afterward, the score of each covered line is computed with one of the SBFL formulas, by default Ochiai. FLACOCO is built in a way that allows the use of different SBFL formulas by providing a standard interface that can be used to integrate custom implementations.

To achieve the final result, FLACOCO uses an Abstract Syntax Tree (AST) Bridge or a File Exporter, depending on the user interface initially selected. The AST Bridge provides, via the

API, an annotated AST with suspicious information. This allows easier integration with tools that work at the AST level, for example, repair frameworks such as Astor [35]. The File Exporter can export the Fault Localization results to formats such as JavaScript Object Notation (JSON) and Comma Separated Values (CSV), as well as through custom exporters.

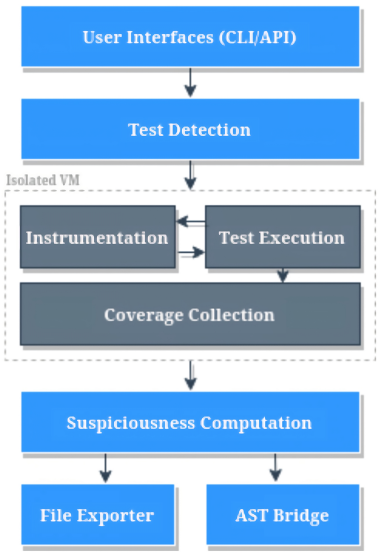


Figure 3.1: FLACOCO's Architecture [2]

FLACOCO has been tested in some experiments. In the original publication [2], two experiments were replicated. The first used the Nopol repair system replicating the Xuan et al. experiment [37] previously mentioned. Originally tested with GZoltar [12], FLACOCO found all of the 17 original patched bugs that GZoltar also found, showing the support for exceptions on three exception bugs detected.

The second experiment of the original publication replicated the work of Martinez et al. [35] while testing Astor, another automatic program repair library that implements different repair approaches. The experiment focused on the jGenProg and jKali approaches. The first is a redundancy-based repair approach, synthesizing candidate fixes from existing code in the system under repair. The last performs an exhaustive search on the code space to be repaired and tries to implement its operators. Both use fault localization techniques to focus on the statements of higher suspiciousness of bugs. By replacing GZoltar, the Fault Localization framework originally used by Martinez et al., with FLACOCO, it was concluded that in both approaches, Astor was able to repair 26 bugs, minus three that in the original experiment with GZoltar. However, Silva et al. tried GZoltar as the Fault Localization framework, reaching the same 26 bugs repaired as FLACOCO. It is thought that the aging of the bugs in Defects4J [41] is the reason for this difference.

3.1.3 Jaguar

Jaguar (JAva coveraGe faUlt locAlization Ranking) is an open-source Fault Localization tool oriented for Java language, which is available not only as an Eclipse Integrated Development En-

vironment (IDE) [42] plug-in but also as a command line tool [3]. It uses SBFL techniques to indicate faulty code excerpts.

Most SBFL techniques use only control-flow spectra, which contain, for example, statements and branch coverage. Data-flow spectra, an increment on control-flow that considers variables definition and their usage, has been suggested by a few studies [3] to perform better than control-flow spectra. However, since it is much more costly than the previous, there is not much application in most of the SBFL studies. Jaguar can implement both data-flow (which also includes the control-flow) and control-flow spectra with an affordable execution cost.

In Fig. 3.2, it is possible to see that Jaguar is composed of a Runner, which collects and generates lists of suspicious program elements and a Viewer, which presents the visual information for debugging.

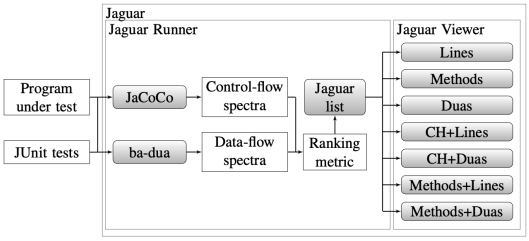


Figure 3.2: Jaguar's Architecture [3]

To collect the control-flow spectrum, Jaguar Runner uses the JaCoCo coverage library [38]. However, when collecting data-flow spectrum, the ba-dua coverage tool[3] is used. Ba-dua uses the Bitwise Algorithm [43] to track variables definition-use associations (duas) at a low execution cost by using inexpensive data structures, only tracking duas that are potentially covered at each node visited during program execution.

Unit tests of the program under test are executed to collect spectra data for each element covered during the test executions. Depending on the coverage tool used, classes, methods, and then lines or duas are iterated. The suspiciousness score of each line or dua is then calculated according to the chosen ranking metric. Ten ranking metrics are available for the calculation: Debugging strategy based on Requirements of Testing (DRT), Jaccard, Kulczynski2, McCon, Minus, Ochiai, O^p, Tarantula, Wong3, and Zoltar [3].

Since each metric has a range of suspiciousness scores, all the calculations are normalized between 0 and 1. These suspiciousness scores can be given to packages, classes, and methods based on the suspiciousness score of the lines or duas. Each package, class, and method score is the highest of their internal lines or duas.

The Jaguar Runner output is a list of packages, classes, methods, and lines or duas and their respective suspiciousness scores as in the example of the Listing 3.1.

[3] https://github.com/saeg/ba-dua

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<HierarchicalFaultClassification metric="OCHIAI" requirementType="LINE" timeSpent="11741">
    <package name="de.susebox.java.util" position="1" susp="1.0">
        <class line="105" name="de.susebox.java.util.AbstractTokenizer" position="1" susp="1.0">
            <method id="43" line="837" name="addKeyword(String, Object, int)" position="1" susp="1.0">
                <requirement line="837" cef="18" cep="1" cnf="0" cnp="102" position="1" susp="1.0"/>
                <requirement line="839" cef="18" cep="1" cnf="0" cnp="102" position="1" susp="1.0"/>
                <requirement line="866" cef="18" cep="1" cnf="0" cnp="102" position="1" susp="1.0"/>
                <requirement line="867" cef="18" cep="1" cnf="0" cnp="102" position="5" susp="0.97"/>
                <requirement line="863" cef="18" cep="1" cnf="0" cnp="102" position="5" susp="0.97"/>
                <requirement line="864" cef="18" cep="1" cnf="0" cnp="102" position="5" susp="0.97"/>
            </method>
            <method id="71" line="1555" name="isKeyword(int, int)" position="2" susp="0.98">
                <requirement line="1561" cef="18" cep="0" cnf="0" cp="103" position="4" susp="0.98"/>
                <requirement line="1565" cef="18" cep="0" cnf="0" cp="103" position="4" susp="0.98"/>
                <requirement line="1564" cef="18" cep="0" cnf="0" cnp="103" position="4" susp="0.98"/>
                <requirement line="1558" cef="18" cep="68" cnf="0" cnp="35" position="171" susp="0.45"/>
                <requirement line="1555" cef="18" cep="68" enf="0" cp="35" position="171" susp="0.45"/>
                <requirement line="1569" cef="1" cep="68" cnf="17" cnp="35" position="450" susp="0.02"/>
            </method>
            <method id="7" line="234" name="addString(String, String, String)" position="5" susp="0.78">
                <requirement line="235" cef="17" cep="9" cnf="1" cnp="94" position="20" susp="0.78"/>
                <requirement line="234" cef="17" cep="9" cnf="1" cnp="94" position="20" susp="0.78"/>
            </method>
```

Listing 3.1: Excerpt of a Jaguar's XML report file [3]

The Jaguar Viewer (example in Fig. 3.3) provides visual information for the output lists from the Jaguar Runner within the Eclipse IDE. The color of an element presented is shown based on its suspiciousness score: $red > 0.75$, $0.75 > orange \geq 0.5$, $0.5 > yellow \geq 0.25$, *and* $green < 0.25$

Users can click on a program element to go to its source code and filter by a source code term in a text search box and/or by suspiciousness score using a slider.

Figure 3.3: Jaguar's Viewer with a method list [3]

With the publication of Jaguar, Ribeiro et al. [3] assessed the tool regarding the effectiveness and efficiency by using the Defects4J database [41] with programs whose sizes vary from 22 to 96 KLOC and test suites that vary from 2k to 4k test cases. The results show that, although Jaguar could not detect all faults, using data-flow spectrum ranks more faults among the top entities compared to control-flow. This happened on the top 10, top 20, and top 30 entities with more suspiciousness of bugs, except for the top 5, where control-flow spectrum was more effective. Overall, out of 173 existing faults, by looking at the top 30 entities, Jaguar found 94 faults using data-flow spectrum and 80 using control-flow spectrum. The use of Jaguar with control-flow spectrum increased the overall execution time, compared to the execution of the basic tests but,

Table 3.1: Java-based SBFL tools comparison based on [8]

Tool	GZoltar	FLACOCO	Jaguar
Open-source	Yes	Yes	Yes
First Release	2012	2021	2017
Latest Official Release	Mar 2022	May 2022	Apr 2018
SBFL supported formulas	16	1	10
SBFL formula extensibility	Yes	Yes	Yes
SBFL spectrum	Control-Flow	Control-Flow	Control-Flow & Data-Flow
Type	CLI tool, Ant & Maven plugin & VSCode extension	CLI tool & Java API	Eclipse IDE plugin & CLI tool
Coverage Collection	JUnit & Javassist	JaCoCo	JaCoCo & ba-dua
Modularity Options	Line	Line	Line, Dua
Programs tested against	Apache Commons Math	Apache Commons Math	Defects4j
Testing Evaluation Criteria	Statement ranked as faulty[a]	Statement ranked as faulty[a]	Faulty statements in Top-n
Faults tested	Single	Single	Single
Largest codebase tested	79 KLOC	79 KLOC	96 KLOC
User study	Yes	Yes	Yes

[a]Inferred from the study of Nopol repair system programs [37]

in all cases, it took less time to execute than data-flow spectrum. The difference varies from 11 seconds to 250 seconds, depending on the program tested.

As far as user assessment, a study was also performed to compare the effectiveness of debugging tasks with and without Jaguar. Out of 26 participants, 12 found a fault using Jaguar while only 5 found a fault without using it. The tool was also evaluated by most as useful and easy to use.

3.1.4 Other Tools

While not as popular, there are other tools available in the Java ecosystem that will not have as much detail in this document. Falcon [44] and Vida [45] are tools that emerged at the start of the previous decade and have remained stagnant without much development. iFL4Eclipse[4] although it is in constant development since it is only an Eclipse [42] plug-in it cannot be considered for this work as it could not be executed in continuous integration systems.

3.1.5 Tools Comparison

Based on the study by Archana and Ashutosh Agarwal [8], Table 3.1 was prepared, considering the information described throughout this section.

Although some information has been obtained regarding the tools, the lack of rigorous tests is notorious, mainly regarding scalability. Furthermore, most of the test results came either from the original publication work on the tool or from work not directly related to the use of the tool. This shows a significant deficit of independent testing.

Real-life systems can also have single, multiple, and concurrent faults. The empirical evaluation of the majority of the SBFL tools has been done with a single at a time.

Regarding the three main tools, they differ from each other in some points. Although all tools support extension in the SBFL formulas used, not all come out of the box with a wide variety. In this regard, GZoltar supports a greater amount without additional work.

[4]https://github.com/InteractiveFaultLocalization/iFL4Eclipse

Jaguar stands out from the rest by enabling the use of data-flow spectrum in an optimized way, which proved to be positive in the experiments carried out since out of 173 faults analyzed, 14 more were found using this method compared to control-flow spectrum. The disadvantage is that it can increase execution time by up to 250 seconds.

Both FLACOCO and Jaguar, theoretically, are easier to support new versions of Java since they do the instrumentation based on JaCoCo, a library in constant update. However, there is a significant difference with this architectural option. Instead of placing a probe at the beginning of each line and executing the test cases in isolation, as GZoltar does, when using JaCoCo, these probes are placed at different positions in the program for better performance. In case of an exception, this can lead to some covered lines being missing. Only FLACOCO mentioned a defense mechanism that might address this problem, so this is likely an issue with Jaguar. Furthermore, if it becomes necessary to change how instrumentation occurs in the future, it is much easier to do so in GZoltar.

Kochhar *et al.* [14] performed an empirical study involving 386 practitioners from more than 30 countries and various companies spread across the globe to get to know their expectations of research in Fault Localization, in particular, the number of factors that impact their willingness to adopt Fault Localization techniques. From this study, it is possible to conclude that most practitioners:

- Value the importance of Fault Localization
- Agree on the availability of some debugging data assumed by most of the Fault Localization studies (single failing test case, passing test cases, multiple failing test cases, etc.)
- Prefer granularity level to pinpoint bugs at methods, blocks, and statements instead of classes and components
- Agree that the faulty program elements must appear at the top 5 on the list of suspiciousness
- Want the techniques to have a minimum success rate of 75%
- Want the techniques to be able to scale to 100 KLOC
- Want it to finish its computation in less than a minute

With the data obtained concerning these three tools, the execution time they take to obtain the necessary data is in doubt. However, considering that the computation is performed after the execution and instrumentation of the tests, taking into account the formulas used by the SBFL, it is expected to take less than the required minute.

Regarding the other requirements, all the highlighted tools are close to reaching 100 KLOC in tests and, depending on the program under analysis, achieve success rates of even greater than 75%, fulfilling the rest of the requirements.

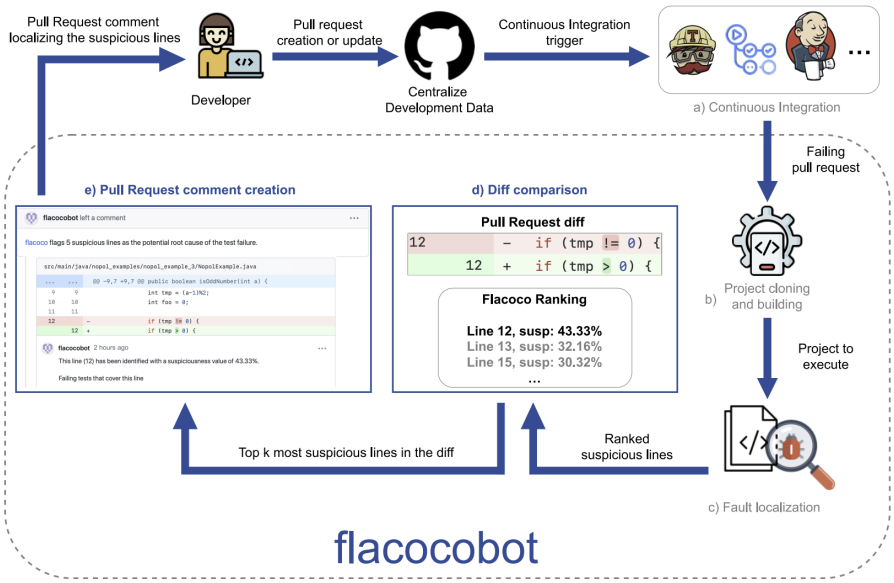


Figure 3.4: Overview of the integration of FLACOCOBOT [2]

3.2 Bots in Software Fault Localization

Bots, automated software agents, can potentially assist developers and testers in various software engineering tasks, including fault localization. Thus, it is natural that bots that use existing SBFL tools have emerged. This section presents the only example of this type of Java tool.

3.2.1 FLACOCOBOT

FLACOCOBOT [2] is a bot that uses the FLACOCO tool to perform fault localization in Java programs. As shown in Figure 3.4, an event triggers the CI pipeline when developers create or update pull requests on a centralized development platform like GitHub. This pipeline runs the tests to check if the pull request does not introduce a regression. In case a failing build is found, FLACOCOBOT, which autonomously scans the status of the pipeline from a list of projects, will intervene by cloning and building the project, execute the fault localization with the FLACOCO tool and finally, post the results as a comment on the pull request. All these previous steps are performed in a Docker container, computing the suspiciousness values for the lines.

Instead of using all the information obtained, FLACOCOBOT just points out some of the pull request diff lines with the highest suspicion value. The idea is to avoid context changes while maximizing the likelihood of being useful. In this way, and as it is possible to see in Figure 3.4 for the GitHub platform, a comment is made in the pull request with a message reporting all the selected lines, as well as the tests that failed and that covered them.

This tool has been tested on ten open-source Java projects available on GitHub [2]. Of these projects, seven had more than 100 KLOC, yet FLACOCOBOT managed to find suspicious lines of code in at least one pull request in nine of these ten projects.

3.3 Summary

In this chapter, the most popular SBFL tools for Java were introduced. A comparison between the tools discovered in the literature was conducted, allowing to understand the strengths and weaknesses of each one, as well as the expectations of its potential users.

The only example of a bot for CI environments that uses these SBFL tools for Java was also presented, showing its architecture and functioning as a reference for the implementation of the tool proposed in this work. The next chapter will detail the proposal for this solution as well as its implementation.

Chapter 4

GZoltar Automatic Debugging for GitHub Actions

As discussed in previous chapters, manually debugging software programs can be time-consuming and costly, requiring significant effort from developers. With software projects becoming more complex and involving larger teams, the challenges of manual debugging only increase. While automated debugging techniques have been proposed as a solution to this problem, they have yet to gain widespread adoption due to their lack of integration with existing systems.

There is evidence of the need for a tool [14] that can automate the application of these techniques in a continuous integration environment. This tool would make it easier for developers to detect faults in their code, reducing the time and effort required for debugging.

A tool that combines the strength of automated debugging techniques with a report that is simple to understand is presented as a solution to this issue. This tool would be integrated directly into the code hosting service used by the developers, making it easier to analyze and apply these techniques. By creating it in a centralized platform for managing all code-related information, this tool would optimize the debugging process, reducing the time spent on fault detection and providing a more streamlined workflow for developers. With this tool, developers could spend more time writing and improving code rather than on the cumbersome and time-intensive task of manual debugging.

4.1 System Specification

For the proposed tool, SBFL techniques are going to be considered to automate the debugging process. As previously stated, these techniques not only have low execution costs [15] but have also been widely used in the field of Software Fault Localization, appearing in 41% of research publications [1].

Some of the SBFL tools that appeared stood out in the Java community. As previously stated, GZoltar can be considered one of the most performing tools available, and due to the involvement of the work's supervisors in its conception, it stands to reason that it is going to be used.

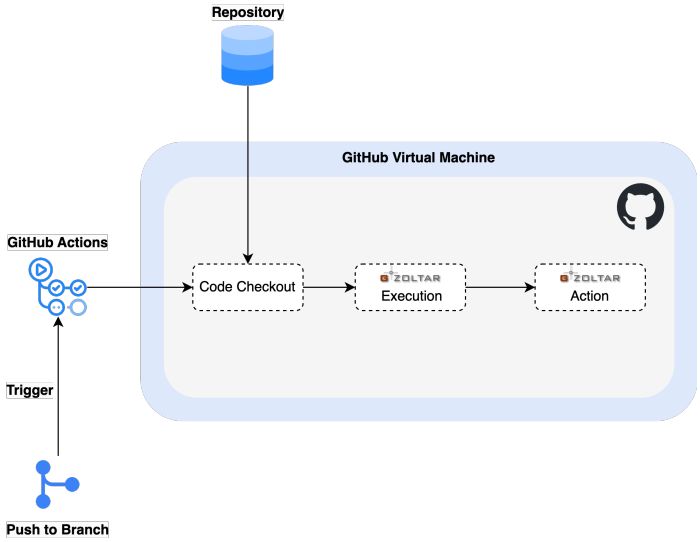


Figure 4.1: Solution's Architecture

To automate the execution of this solution in a continuous integration environment, it is required first to determine which environment is most suited for this purpose.

Among the choices presented in Section 2.3, GitHub Actions has one of its main advantage, its tight integration with GitHub. This allows one to easily trigger a workflow based on events such as pull requests or pushes to a repository and to view the status of a workflow directly in a GitHub pull request or commit. Furthermore, its pricing model offers a generous free tier that includes up to 2,000 workflow minutes per month in case of executions in private repositories or unlimited workflow minutes in case of executions in public repositories [46], such as open-source projects, without the need for server configuration, unlike other CI tools. This makes it an attractive option for many developers.

GitHub Actions also provides a large number of pre-built actions for many tasks, which can save a lot of time and effort when setting up a workflow. These actions, made available on the GitHub Marketplace [47], are a perfect example of the power of open-source software with over 10,000 created by the community that can be used by anyone [48], free of charge. This place is perfect for a tool that aims to automate the debugging process just like the one it is being proposed, as it can be easily integrated with developers' existing workflow.

GitHub is the most widely used code hosting service, with 94 million users and 263 million automated jobs run on GitHub Actions every month as of 2022 [49]. As a result, it is the most suitable environment to reach a higher number of developers with little additional configuration.

Considering the architecture present in Figure 4.1, the proposed tool will be implemented as a GitHub Action. It will be integrated with the GZoltar tool, which will be used to perform the SBFL techniques with the analysis of the test execution on the pipeline. Therefore, it is expected that this action, in addition to being executed after the analysis of GZoltar, will be executed in projects already with the presence of tests that can be executed in a CI pipeline.

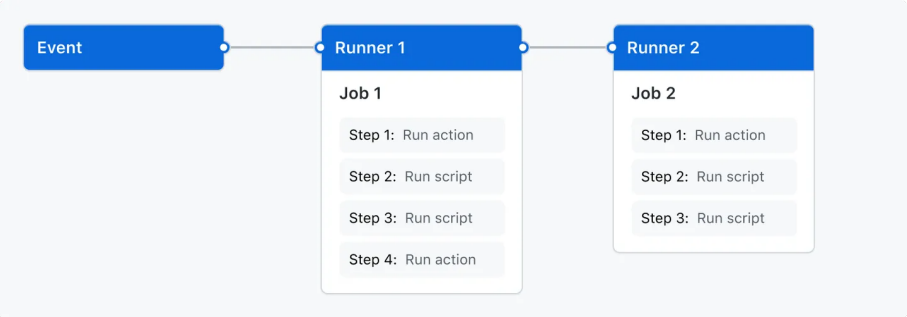


Figure 4.2: Github Actions Components [4]

GitHub Actions goes beyond DevOps and allows running workflows when other events happen in a repository. The main components of GitHub Actions are represented in Figure 4.2. With this architecture, it is possible, for example, to run a workflow to automatically add comments to a commit whenever someone pushes to a branch in a repository. This can complement the GZoltar run and generate summarized and simple-to-understand results.

A workflow has one or more jobs that can execute sequentially or concurrently. Each job will operate in its virtual machine runner or container and will have one or more steps that will either execute a script described or an action. An action is a reusable job that can simplify a workflow. By publishing an action to the GitHub Marketplace, anyone can use it in their workflows.

Workflows are defined by a YAML file contained in a repository that can be run when triggered by:

- **Events** - for example, when someone pushes to a repository or when a pull request is created
- **A schedule** - for example, every day at 12:00
- **Manually** - for example, when someone clicks a button in the GitHub UI

These files are defined in the *.github/workflows* directory in a repository, allowing it to have multiple workflows, each with a different set of tasks. For example, a repository can have a workflow that runs at a specific time or interval, such as every day or every week, to perform routine tasks such as updating dependencies or cleaning up old data, and another workflow that runs when a pull request is created to run the proposed action and add comments to the pull request with the results.

This specification allows different workflows for different branches, such as running the action only in the main branch or running it in all branches but only when a pull request is created. The example in Listing 4.1 shows a workflow that runs when a commit is pushed to a repository - note the property *on* - and that runs a job called *check-bats-version* on a Ubuntu [50] runner with multiple steps:

1. Checkout the repository to the runner using a widely used action by the community *actions/checkout* [51]

2. Setup the Node.js environment using another widely used action *actions/setup-node* [52]
3. Install the *Bats* testing framework [53]
4. Run the Bats version command

```
1   name: learn-github-actions
2   run-name: ${{ github.actor }} is learning GitHub Actions
3   on: [push]
4   jobs:
5     check-bats-version:
6       runs-on: ubuntu-latest
7       steps:
8         - uses: actions/checkout@v3
9         - uses: actions/setup-node@v3
10          with:
11            node-version: '14'
12        - run: npm install -g bats
13        - run: bats -v
```

Listing 4.1: Example of a Workflow YAML file [4]

To make available a pre-built action, it is necessary to create a custom action. These actions can be of the following types:

- **JavaScript action** - a JavaScript file is run directly on the runner
- **Docker container action** - a Docker container that runs on a Linux runner
- **Composite run steps action** - a combination of multiple workflow steps within one action

Since the proposed action analyzes results already prepared by GZoltar, it is implemented as a JavaScript action. This type of action is faster to run on the GitHub runners since it does not need to build and retrieve containers, as it happens with Docker container actions, or to download and extract the action, as it happens with composite run steps actions [54]. Furthermore, multiple packages can speed up the development, allowing interaction with the GitHub API and easily changing the repository, such as the *actions/toolkit* packages [55].

A custom action needs to be declared using an action metadata file. On the listing 4.2, an example of a custom JavaScript action named *Hello World* is shown. This action has a single input called *who-to-greet* that is required and has a default value of *World*. It also has a single output called *time* that will store the time when the action was executed. Finally, it has a single step that runs a JavaScript file called *index.js* using the *Node.js* 16 runtime.

```
1   name: 'Hello World'
```

```
2     description: 'Greet someone and record the time'
3     inputs:
4       who-to-greet:  # id of input
5         description: 'Who to greet'
6         required: true
7         default: 'World'
8     outputs:
9       time: # id of output
10        description: 'The time we greeted you'
11    runs:
12      using: 'node16'
13      main: 'index.js'
```

Listing 4.2: Example of an Action metadata YAML file [9]

With an action metadata file prepared in a repository, it is only needed to write the JavaScript code the action will execute.

4.2 Implementation

4.2.1 Initial Configuration

A custom action named *GZoltar Automatic Debugging*[1] was created and published on the GitHub Actions Marketplace [56]. The first configuration batch consisted of maintenance and help tools for the programmers, such as static code analysis or static typing for *JavaScript*.

ESLint [57], a popular *JavaScript* linter, was used to analyze the code for potential errors, best practices, and other issues. It was configured to use the *eslint:recommended* ruleset, which is a set of rules considered useful and maintained by the *ESLint* team. This was done using the configuration file *.eslintrc.json*, containing the set of rules and configuration options that define how *ESLint* should analyze your code

Prettier [58], a popular code formatter, was used to format the code according to a set of rules. It was configured based on the *actions/checkout* [51], one of the most used GitHub actions created by the own platform team, allowing the code to be formatted accordingly with the actions standard. This was done using the configuration file *.prettierrc.json*, containing the rules and configuration options that define how *Prettier* should format your code.

TypeScript [59], a popular superset of *JavaScript* that adds static typing and improves the code readability, was used. This was done using the configuration file *tsconfig.json*, containing the set of rules and configuration options that define how *TypeScript* should compile your code with the default configuration of the *Node.js 16* as the basis. The *strict* flag enabled a wide range of type-checking behavior, resulting in stronger program correctness guarantees.

[1] https://github.com/GZoltar/gzoltar-github-action

Since the GitHub Actions runner and metadata file needs a single entry point, a file named *index.js* was used for that purpose. This file contains all the logic inside the TypeScript files that were packaged into a single file using *ncc* [60]. Standing for *Node.js Compiler Collection*, this tool allows deploying *Node.js* applications to environments where it may be difficult or impossible to install dependencies, such as serverless functions, containerized environments, or the GitHub Actions runner when configured as a *JavaScript* action. As a result, the code could be kept organized with logical partition by multiple files while still allowing the constructed action to work.

All these tools could be executed with the help of the *scripts* defined on the *package.json*, a metadata file used in *Node.js* projects to define various aspects of the project, such as its name, version, dependencies, and scripts. This allowed the automation of everyday tasks in the development process, making it easier to run tests, build the project, and perform other tasks.

Upon completion of the previous settings, a workflow was set up on the repository. Its function is to ensure that the packaged version always reflects the logic of the files as it is at the time of use. This workflow was obtained through the *actions/checkout* [51] and placed in the *dist-checker.yml* file inside the folder corresponding to workflows, *.github/workflows*. The main objective is to execute the package of the *TypeScript* files using the *ncc* tool and compare the obtained file with the entry file *index.js* present in the repository, warning the developer whenever the two files do not match.

4.2.2 Action Definition

As seen previously, it is necessary to define its various properties in a metadata file to create an action. The *GZoltar Automatic Debugging* action was defined, giving a name, description, several inputs, and a run definition stating that it will use the *Node.js 16* runtime and the *dist/index.js* file as an entry point. No output properties were defined since the purpose of the action is to summarize the results of GZoltar through comments in Pull Requests/Commits and not precisely to make them available for later use in CI pipelines.

Regarding the inputs, the user has got the following options:

- ***token*** - Personal access token (PAT) used to make actions on the repository, such as creating comments on PRs/Commits using the GitHub API. It is recommended to use a service account with the least permissions necessary, and when generating a new PAT, select the least scopes necessary. [61]
- ***build-path*** - Path to the build/target directory containing the GZoltar results. Default: *'/build'*
- ***serialized-coverage-file-path*** - Path to the file containing the serialized file with the coverage collected by GZoltar. Example: *'/build/gzoltar.ser'*. (Optional)
- ***test-cases-file-path*** - Path to the file containing a list of all test cases identified by GZoltar. Example: *'/build/sfl/txt/tests.csv'*. (Optional)

- ***spectra-file-path*** - Path to the file containing a list of all lines of code identified by GZoltar (one per row) of all classes under test. Example: *'/build/sfl/txt/spectra.csv'*. (Optional)
- ***matrix-file-path*** - Path to the file containing a binary coverage matrix produced by GZoltar. Example: *'/build/sfl/txt/matrix.txt'*. (Optional)
- ***statistics-file-path*** - Path to the file containing statistics information of the ranking produced by GZoltar. Example: *'/build/sfl/txt/statistics.csv'*. (Optional)
- ***ranking-files-paths*** - Path to each SBFL ranking algorithm file. Example: *'[/build/sfl/txt/ochiai.ranking.csv]'*. (Optional)
- ***sfl-ranking*** - List of the SBFL ranking algorithms to use separated by commas. (Remember that each algorithm needs to have a fault localization report file in the ranking-files-path directory with its name, i.e., ochiai.ranking.csv.) Default: *'[ochiai]'*
- ***sfl-threshold*** - Line suspiciousness threshold to trigger a warning and show results. A threshold is needed for each SBFL ranking algorithm used. Default: *'[0.5]'*
- ***sfl-ranking-order*** - Ranking algorithm to order table results by suspiciousness in descending order. Default: *'ochiai'*
- ***diff-comments-code-block*** - Indicates if comments displayed on files with suspicious lines in the diff are grouped by code block instead of each line. Default: *true*
- ***upload-artifacts*** - Indicates whether to upload the GZoltar results as an artifact. Default: *false*

Each option is going to be further explained later.

4.2.3 Code Structure

Based on the structure of the *actions/checkout* action, the code was written inside a folder named *src*. The only exception to this are the configuration files, the GitHub Workflow to ensure the latest code is packaged, and the packaged file containing all the code placed on *dist/index.js*.

The *src* folder contains the following files:

- ***dataProcessingHelper.ts*** - Contains the logic to process the data produced by GZoltar and gathered by the action. Most of the logic is related to creating the strings used on the comments for the Pull Requests/Commits.
- ***fileParser.ts*** - Contains the logic to parse the files produced by GZoltar and gather the data the action will use. It gathers information on the test cases, the coverage matrix, the ranking algorithms, etc. It also keeps track of the files exercised by test cases that will be commented on the Pull Requests/Commits and the files containing the HTML reports generated by GZoltar.

- ***fsHelper.ts*** - Contains several functions that allow to read and search files from the file system.
- ***githubActionsHelper.ts*** - Contains several functions that allow interaction with the GitHub API. It is used to create comments on Pull Requests/Commits, get the diff between two branches, and upload artifacts.
- ***inputHelper.ts*** - Contains the logic to parse the Action inputs and validate them, giving detailed errors in case of invalid inputs.
- ***main.ts*** - Contains the entrypoint logic of the action. It calls the FileParser and other helpers and orchestrates the action execution.
- ***stateHelper.ts*** - Contains several variables and functions that allows the action to keep track of the current context. It is used to keep track of the current commit, the current repository, if the action is being called on a Pull Request, etc.

In addition to these files, several types are also defined in the *src/types* folder. These types are used to define the data structure that will be used in the action, such as the information on the test cases, the coverage matrix, the ranking algorithms, etc. This way, it is possible to maximize the use of *TypeScript* and its static typing for *JavaScript*.

The available types are the following:

- ***inputs.ts*** - Interface used to define the structure of the Action inputs, containing all the options available previously discussed.
- ***fileOnDiff.ts*** - Interface used to define the files present on diff between commits. It contains the path of the file and the changed lines (a list of instances of the *DiffChangedLines* Interface).
- ***diffChangedLines.ts*** - Interface used to define the lines changed on diff between commits. It contains a batch of changed lines of a file on the diff with the indication of the line numbers of the start and end of this batch, allowing the action to understand if a specific line number is contained in any of the batches of changed lines of a file.
- ***sourceCodeFile.ts*** - Interface used to define the structure of the source code files. An instance is supposed to represent a file exercised by at least a test case. It contains the path of the file, the name of the file, and the *Java* package name to which this file belongs.
- ***sourceCodeLine.ts*** - Interface used to define the structure of the source code lines. An instance is supposed to represent a line of a file exercised by at least a test case. It contains the line number, the method to which it belongs (using an instance of the *SourceCodeMethod* Interface), and the suspiciousness scores of this line for each SBFL ranking algorithm used (using a list of instances of the *AlgorithmSuspiciousness* Interface).

- ***sourceCodeMethod.ts*** - Interface used to define the structure of the source code methods. An instance is supposed to represent a file method exercised by at least a test case. It contains the name of the method, the file to which it belongs (using an instance of the *SourceCodeFile* Interface), and a list of the parameters of the method.

- ***algorithmSuspiciousness.ts*** - Interface used to define the structure of the suspiciousness scores of a line for an SBFL ranking algorithm. It contains the name of the algorithm and the suspiciousness score.

- ***testCase.ts*** - Interface used to define the structure of the test cases. An instance is supposed to represent a test case. It contains the name of the test case, an indication if the test passed or failed, the runtime of the test in nanoseconds, the stack trace (if applied), and the information on coverage of the test case for several source code lines (using a list of instances of the *testCaseCoverage* Interface).

- ***testCaseLineCoverage.ts*** - Interface used to define the structure of the coverage of a test case for a source code line. It contains the line instance (using the *sourceCodeLine.ts* Interface) and an indication if the test case covered the line.

- ***statistic.ts*** - Interface used to define the statistics information of the SBFL rankings used by GZoltar. Each instance is supposed to contain the name of the ranking algorithm, a metric name, and the metric value. Examples of metrics supported by the GZoltar for these statistics are ambiguity and entropy.

Finally, there is a directory where assets *src/assets* are kept, such as images representing the severity of the suspiciousness score through the colors red, orange, yellow, and green. These images are used in the comments made by the action on the Pull Requests/Commits.

4.2.4 External Packages

Apart from external packages related to the initial configuration 4.2.1, such as *ncc* [60], the only packages that were used were part of *actions/toolkit* [55]. *@actions/artifact*, *@actions/core* and *@actions/github* were needed with the following functions:

- ***@actions/artifact*** - Allows you to easily create, upload, and download artifacts during a workflow. In this context, it is used to upload the HTML/Text reports generated by GZoltar as artifacts of the action.

- ***@actions/core*** - Offers essential functionality for interacting with the GitHub Actions runtime and workflow environment. In this context, it is used to get the inputs of the action, set the outputs of the action, and set the status of the action.

- ***@actions/github*** - Allows you to interact with the GitHub platform and access information related to the repository, pull requests, issues, and other GitHub entities within a workflow.

In this context, it is used to get the diff between commits, create comments on Pull Requests/Commits and get the information on the current repository.

4.2.5 *Modus operandi*

4.2.5.1 Initial Parsing of GZoltar report files

The action starts by interpreting and validating the inputs using the *@actions/core* package, placing them in *TypeScript* objects. Validation involves checking that required inputs are present and other aspects inherent to the action's logic, such as whether the number of elements in the input *sfl-ranking* and *sfl-threshold* are equal. The action ends with an error message if the inputs are invalid.

Having all the inputs correct, an instance of the *FileParser* class is created, which will contain all the information processed by GZoltar in objects with the abovementioned types. The necessary inputs are passed to this class which starts by parsing the Test Cases file named *tests.csv*. If no file path is passed in the inputs, the program looks in the *build* directory for it. This happens for this file and most of the files needed for parsing. According to the file's content, several instances of the type *ITestCase* will be created containing the name of the test, an indication if the test passed, the execution time in nanoseconds, and the stack trace, if any.

Next, the file containing the spectra called *spectra.csv* is parsed. This file contains a list of all lines of code identified by GZoltar. In this way, instances of the types *ISourceCodeFile*, *ISourceCodeMethod* and *ISourceCodeLine* are created, which allows storing all types of information of the lines of code considered in the instrumentation of the code as well as the method of a line of code and the file of that method. It should be noted that in this process, the entire source code is searched to see if a file exists so that it can later be referenced directly on GitHub.

As a core part of this processing, the files containing the suspiciousness values for each SBFL ranking algorithm are parsed. During this process, it is verified that all the lines of code and consequent previously processed methods and files that are referenced exist in the system, and the list of suspiciousness metrics that each instance of *ISourceCodeLine* has is updated with a given value for a given algorithm.

To finish processing the primary data, the binary coverage matrix is processed. The file that contains this matrix is called *matrix.txt* and, when processing it, it is possible to indicate in the instances of type *ITestCase* whether each test passed or not and whether a given line, represented by an instance of *ISourceCodeLine* previously created, was covered by the test or not.

Although less critical for the execution, in the end, the statistics file containing the serialized coverage information used by GZoltar to generate its reports and the files with the reports in GZoltar's HTML format, if any, are also parsed. For the last two, only their paths are saved for later upload if the user wants to.

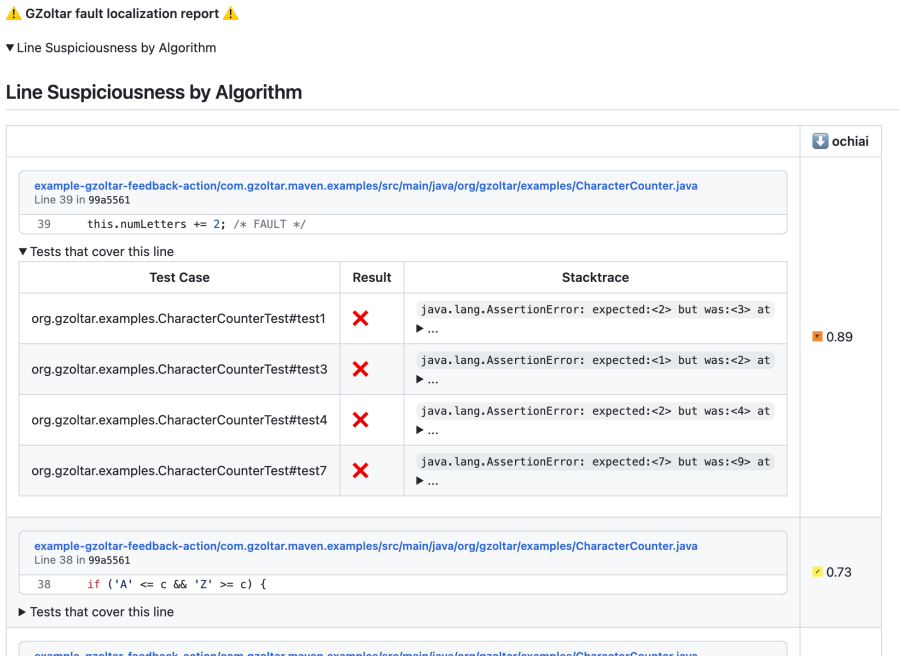


Figure 4.3: Example of a table with the line suspiciousness by algorithm generated by the action

4.2.5.2 Creating Comments on GitHub

With all of the data ready, the functions in the helper created to interact with the GitHub API are used. The entry point to create the comments is the function *createCommitPRCommentLineSuspiciousnessThreshold*, which, based on the instances of *ISourceCodeLine* previously created, filters the lines of codes that have a suspiciousness score equal to or above the threshold defined by the user, for the SBFL ranking algorithms also defined by the user. Since an option to define the order of the lines of code presented based on the suspiciousness score of an algorithm is required on the inputs and given to the program, the lines are ordered according to the algorithm chosen from the lines with more suspiciousness to the less.

The main report is then generated, and the comment on the commit or pull request is created. This report contains two tables:

- **Line Suspiciousness by Algorithm** - Contains the lines of code and the suspiciousness score for each algorithm. For each line, not only a preview with the code is shown, as well as the method and file to which it belongs on the repository, but also the coverage information for each test case that covers that line. This information is presented in another table with the indication of the test case's name, if it passed, and the stack trace, if any. Figure 4.3 shows an example of this table.

- **Lines Code Block Suspiciousness by Algorithm** - Contains a block of lines of code of up to 11 lines in a row with the suspiciousness score for each algorithm and line of the block. As in the previous table, the view is of a preview of the code and the method and file to which it belongs on the repository. Figure 4.4 shows an example of this table.

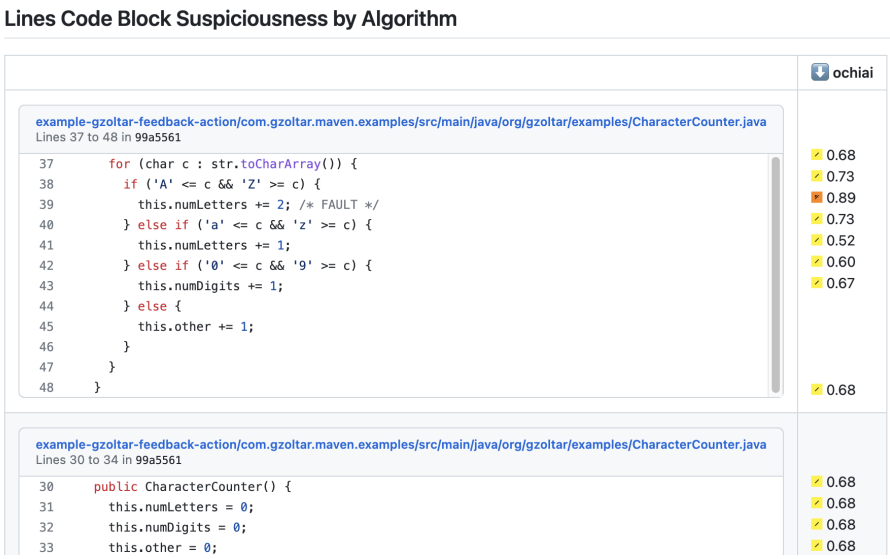


Figure 4.4: Example of a table with the lines code block suspiciousness by algorithm generated by the action

For both tables, methods of the *dataProcessingHelper.ts* created to interact with the data are used. These methods allow for generating the table ready to be presented, with data formatted and colors according to the suspiciousness value. Grouping lines by methods and lines in a row, generating the preview link of lines on GitHub, and ensuring the formatting of tables created in *Markdown* [62], the markup language used by GitHub in writing documentation on the platform, are examples of the processing done.

The colors associated with the suspiciousness value are displayed following *Coloradd* [63], a color system that allows color-blind people to distinguish colors, and the following table:

Suspiciousness Value	Color
>= 0.90	
>= 0.75	
>= 0.50	
< 0.50	

Table 4.1: Suspiciousness values and their respective colors

To create the comment, the package *@actions/github* is used to interact with the GitHub API along with the PAT token provided by the user or automatically provided by the default value. The *stateHelper.ts* is used to identify either if the action is running on a commit or a pull request and getting the commit secure hash algorithm (SHA) - an identification of the commit on the Git version control system [64] - or pull request number, respectively. With this information, the comment can be created successfully.

In addition to this main report, it is also tried to create a comment in the commit diff itself (or the last commit in the case of a pull request). For this, the GitHub API is used again to identify the files in the diff that have not been deleted. The SHA of the current commit (or most recent in

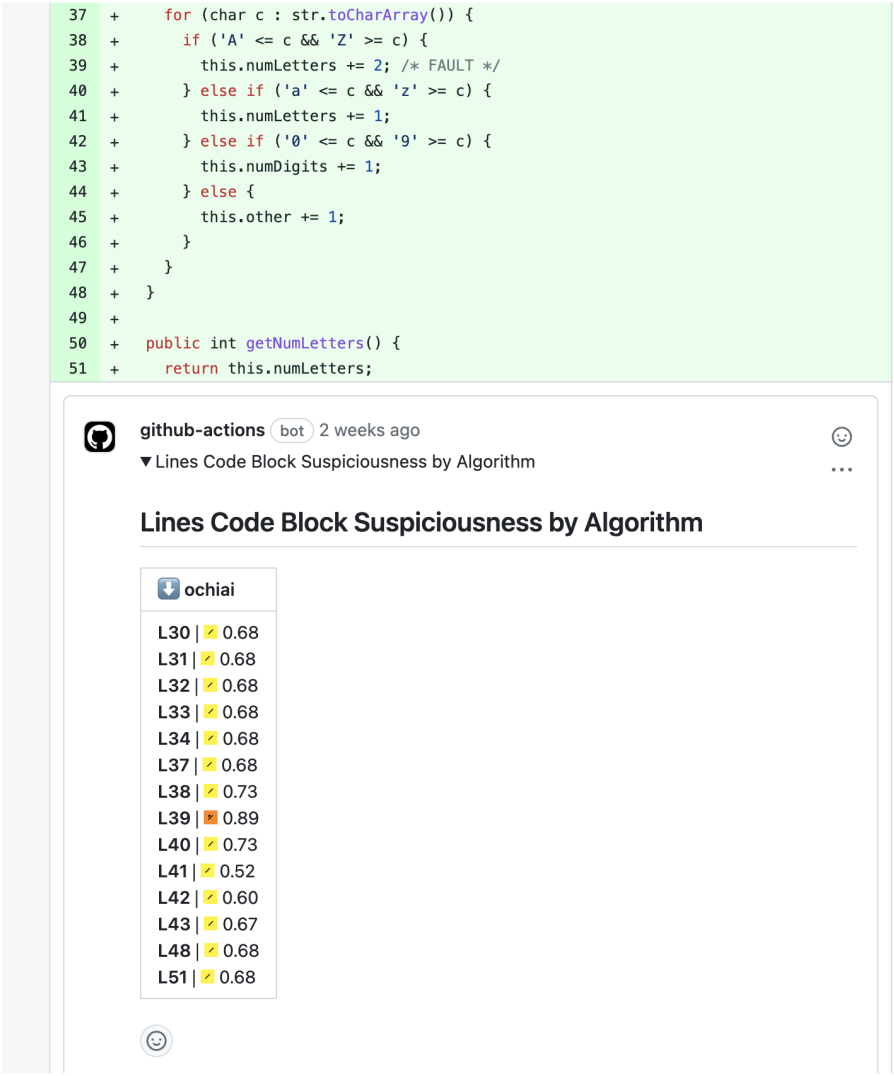


Figure 4.5: Example of a table with the lines code block suspiciousness by algorithm generated by the action in the diff

the case of a pull request) and the previous one (or first of the Pull Request) are used to make a comparison. With this information, the lines present on the files shown on the diff can be filtered, and new tables can be generated.

Depending on the option *diff-comments-code-block* selected by the user on the input, tables representing the Line Suspiciousness by Algorithm or Lines Code Block Suspiciousness by Algorithm can be generated in a similar way to the main report. It should be noted that in the case of tables of the first type, the comment is made directly on the line present in the diff, while in the second type, the comment is made directly on the last line of the block present in the diff. Moreover, the preview of the code is never shown once since it is commenting directly on it. An example of a Lines Code Block Suspiciousness by Algorithm table can be seen in Figure 4.5.

It is important to note that for this process to take place, it is necessary to give additional permissions to the GitHub job with *contents* and *pull-requests* write permissions on the YAML

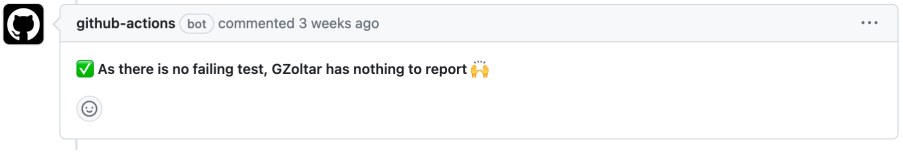


Figure 4.6: Informative message presented to the user when no tests fail

workflow definition. This is needed for the action to be able to create comments on both commits and pull requests.

Evidently, all this processing and content generation can only happen if tests fail, giving the necessary execution information to GZoltar. If this is not verified, only an informative message that no test failed visible in Figure 4.6 is presented instead of the main report.

4.2.5.3 Upload GZoltar's Artifacts

If so desired by the user, all the information generated by GZoltar can be included in the execution of the action for a better grasp of the data. The files to be sent are part of all the references used and saved during the parsing process. This includes the matrix with the binary coverage, the spectra file, and even the HTML reports if these have been generated, among others.

For this purpose, the *@actions/artifact* package is used to send artifacts that will be available on action execution information. These artifacts allow to persist data after a job has been completed and share that data with another job in the same workflow. An artifact is a file or collection of files produced during a workflow run. In this case, a zip containing all directories and files found and related to the information produced by GZoltar is sent. This zip is then available for download in the action execution information, as shown in Figure 4.7. The structure of the unzipped artifact can be seen in Figure 4.8.

4.2.6 Limitations

During the development of the action, two limitations not inherent to the developed action were detected, which could affect its functioning.

4.2.6.1 JUnit version

Although a new version of GZoltar that will allow it to run in Java programs that use the latest major JUnit version 5 is under development [65], currently GZoltar only supports JUnit version 4 or lower. This turns out to be a relatively significant restriction since the last release of JUnit 4 occurred more than two years ago [66], limiting the programs that can be integrated since it is natural for developers to update dependencies.

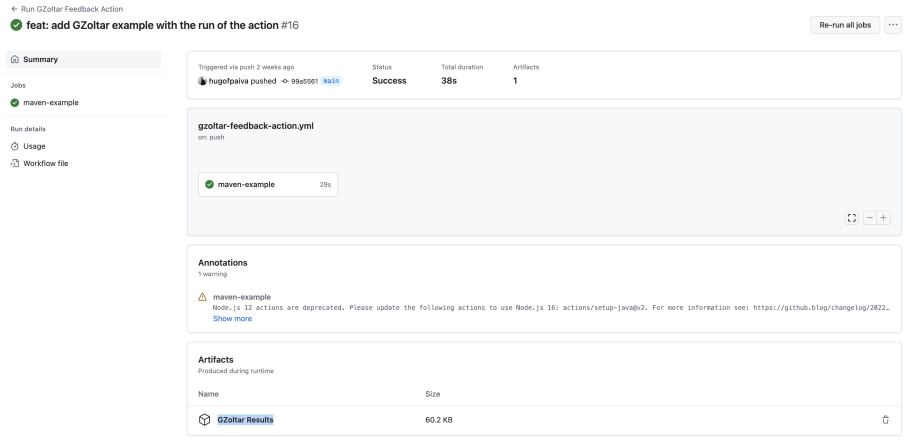


Figure 4.7: Example of the artifact generated by the action

4.2.6.2 Main Comment String Length

GitHub has a limit of 65536 characters for the main comment of a commit/pull request [67]. This limit is not related to the action but to the GitHub API itself. If the limit is exceeded, the action will fail.

The main comment string can get quite long because, in the case of the Line Suspiciousness by Algorithm analysis, the tests that covered a given line are shown, including the stack trace, in case of a test failure. To try to solve this problem, the maximum size of the stack trace was reduced to 300 chars, placing "..." when this is exceeded. If this problem recurs in several environments, it is always possible to make developments to reduce this size or even put it as an action input, but this will result in less information shown to the user.

4.3 Usage Example

In order to present examples of use and carry out an initial test of the developed action, a repository with an example of running the action was created [10]. This repository contains the GZoltar Maven example [68], with the addition of the entire GitHub CI pipeline configuration to run the tests with the generation of GZoltar reports and, afterward, the developed action. Thereby, a possible future user can verify the execution of the action and its integration with GZoltar without having to carry out any initial configuration. Although the example of Maven provided by GZoltar was used, GZoltar can run in any simple Java program through the CLI or, similar to Maven, by using Apache Ant. Both ways also have examples in the GZoltar repository.

The workflow configuration for this example was composed of the Listing 4.3. This workflow is triggered by a push to the main branch or by a manual trigger and is composed of five steps:

1. Checkout the repository
2. Setup Java 8
3. Compile the tests with Maven

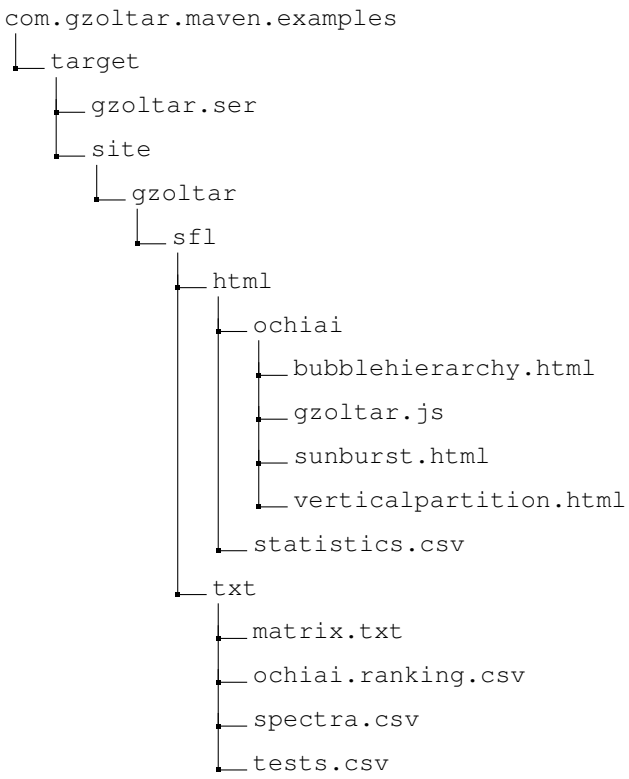


Figure 4.8: Tree structure of an example unzipped artifact

4. Run the tests and GZoltar with Maven
5. Generate the GZoltar report
6. Run the developed action, GZoltar Automatic Debugging for GitHub Actions

```
1  name: Run GZoltar Action
2
3  on:
4    push:
5      branches:
6        - main
7      paths-ignore:
8        - "**.md"
9    workflow_dispatch:
10
11 jobs:
12   maven-example:
13     permissions:
14       contents: write
15       pull-requests: write
```

```
16    runs-on: ubuntu-latest
17    defaults:
18      run:
19        working-directory: ./com.gzoltar.maven.examples
20    steps:
21      - uses: actions/checkout@v3
22
23      - uses: actions/setup-java@v2
24        with:
25          java-version: "8"
26          distribution: "temurin"
27
28      - name: Compile Tests with Maven
29        run: mvn clean test-compile
30
31      - name: Run Tests and GZoltar with Maven
32        run: mvn -P sufire gzoltar:prepare-agent test
33
34      - name: Prepare GZoltar Report
35        run: mvn gzoltar:fl-report
36
37      - name: Run Gzoltar Action
38        uses: GZoltar/gzoltar-github-action@v0.0.1
39        with:
40          build-path: "/com.gzoltar.maven.examples/target"
41          sfl-ranking: "[ochiai]"
42          sfl-threshold: "[0.5]"
43          sfl-ranking-order: "ochiai"
44          upload-artifacts: true
```

Listing 4.3: Workflow YAML file of the example of running the action developed [10]

For the execution of the GZoltar's action, most of the default inputs were used, except for the *build-path* and the *upload-artifacts* inputs. This means that the chosen SBFL algorithm was *ochiai* with a minimum threshold of 0.5 and artifact upload enabled. The *build-path* input was set to the path of the *target* folder, the Maven build directory, where all the information generated by GZoltar is saved.

Once a commit has been made, the result includes a comment with the main report in the commit itself and just a comment in the commit diff of the type Lines Code Block Suspiciousness by Algorithm since all the lines of code with possible bugs were in the same file, next to each other. Furthermore, all GZoltar artifacts detected during the run were uploaded to GitHub. The results are visible in the Figures 4.3, 4.4, 4.5, 4.7 and in the commit itself [2].

[2] Commit with the action execution example

Running the entire action, including GitHub processes to start and shut down a worker, took about 23 seconds, and the action also reported all errors reported by GZoltar.

4.4 Summary

In this chapter, the architectural choices of the developed action were discussed, as well as the tools used for its implementation. All the tools were explained in detail and the functioning of the action itself was fully described.

An example of using the action was also presented as well as the inherent limitations. The next chapter will present the developed evaluation and subsequent discussion of the action.

Chapter 5

Discussion

Several steps were taken to evaluate the developed action to prove its effectiveness and usability to the target audience. The first step was to perform a qualitative evaluation with FLACOCOBOT, a tool with similar objectives. However, even more important, the action was run in a set of open-source repositories in order to evaluate its effectiveness in a real-world scenario.

5.1 Comparison with other tools

5.1.1 FLACOCOBOT

Although the basic concept of FLACOCOBOT is the same as the developed action, some notable differences are summarized in Table 5.1.

One of the most notorious is that FLACOCOBOT is not a GitHub Action but a bot that runs on a server and needs to be configured by the tool developer. This means the tool is not available for any open-source project but only for those the tool developer configures. In addition, the tool is not open-source, so it is impossible to analyze its code and understand how it works.

Regarding filtering the results of the underlying SBFL Java Tool, both work differently. FLACOCOBOT allows selecting the number of lines with the highest suspiciousness to comment. In contrast, GZoltar Automatic Debugging allows setting the threshold of suspiciousness from which the lines are considered on the report.

Both tools allow creating the report in pull requests. FLACOCOBOT, however, does not allow making the report only in commits or directly in the diff. Additionally, it only reports information one line at a time in a more textual form, without the possibility of using a view like the one in the code block mode of GZoltar Automatic Debugging. Moreover, FLACOCOBOT supports one formula, while GZoltar Automatic Debugging supports 16 different formulas.

Finally, FLACOCOBOT does not allow the user to debug the results of the underlying SBFL tool as opposed to GZoltar Automatic Debugging which enables the user to upload the artifacts to the GitHub repository and debug the results.

Table 5.1: Comparison of GZoltar Automatic Debugging for GitHub Actions and FLACOCOBOT

Features	**GZoltar Action**	**FLACOCOBOT**
Allows selection of top k suspicious lines	No	Yes
Allows selection of threshold of suspicious lines	Yes	No
Users can configure it without the assistance of developers	Yes	No
Comment on Pull Request	Yes	Yes
Comment on Commit	Yes	No
Comment in code diff	Yes	No
Code block mode	Yes	No
Formulas supported	16	1
Debug the underlying SBFL tool results	Yes	No

5.2 Running the action in the open-source world

The open-source world represents a collaborative and transparent approach to software development, where the source code is freely accessible and modifiable. This inclusive model has had a transformative impact on technology and society. At the forefront of this movement is GitHub, one of the most trusted platforms to host open-source projects, providing an ecosystem for developers to collaborate.

Open-source projects have become increasingly significant in various industries, offering numerous benefits and driving innovation. One notable example is the widespread adoption of the Linux operating system in the tech industry. Developed as an open-source project, Linux has gained immense popularity and is now used by 100% of the world's top 500 supercomputers and 85% of smartphones [69]. Furthermore, adopting open-source databases, such as MySQL and PostgreSQL, has transformed the industry's data management field, being within the top-4 of popular database management systems [70]. This demonstrates open-source software's critical role in powering major industries' infrastructure.

5.2.1 Methodology

In order to test the developed action in the real world and prove its usability in the industry, the objective of this section is precisely to execute GZoltar and the created action in several open-source programs. The main goal is to verify the action's ability to help detect bugs in real-world programs and to understand the action's impact on the development process of these programs.

For this, all the action configuration work must be minimized to increase the integration possibilities in the program development process. Furthermore, none of the configurations can damage the tests, functioning, or configurations of the program to be integrated.

That said, the following steps will be followed for each program to be integrated:

1. **Configure GZoltar and the action execution in the program using a fork of the program repository** - The minimum of changes should be made in order to disturb the development team as little as possible, increasing the chances of integration

2. **Create an example of the action execution in the program in a previous commit where tests failed** - This will allow the development team to visualize the action operation and understand how it works

3. **Create a pull request for the program repository with the configuration changes** - This will allow the development team to review the changes and understand the impact of the action in the program as well as understand how the action works through the insights that will be given in the pull request description

4. **Gather feedback from the development team on the Pull Request and throughout the development process** - To understand the impact and usefulness of the action in the development process

Bearing in mind that GZoltar was designed to run in Java programs that use JUnit to run unit tests and the limitations previously detailed in section 4.2.6, the range of options of programs to be integrated is thus reduced. In addition, the programs to be integrated must be open-source and hosted on GitHub since the developed action is only available for GitHub Actions.

To select potential programs considering these nuances, a repository with a curated list of Java frameworks, libraries and software with more than 300 contributors and 30,000 stars on the GitHub platform [71] was used. Thus, it was possible to analyze multiple programs with the following requirements:

- **Open-source** - The programs must be open-source and available on GitHub since the developed action is only available for GitHub Actions

- **Java programs** - The programs must be written in Java since GZoltar only works with Java programs

- **Unit Tests** - The programs must have unit tests since GZoltar needs to run unit tests to gather the coverage information

- **JUnit** - The programs must use JUnit with version 4 or lower to run unit tests since GZoltar has that limitation

- **CI Pipeline** - The programs must have a CI pipeline with tests configured using GitHub Actions since the developed action needs to be executed in that context, and developers are expected to pay attention to it

- **Maven or Ant** - The programs must use Maven or Ant to manage the project since GZoltar does not support Gradle [72]

- **Activity** - The programs must have recent activity since feedback from the development team is needed. Any project that has not had any activity in the 2023 year was excluded
- **Complexity** - The programs should not be extremely complex since there are limited deadlines within the scope of this academic work. Projects with five or more modules were excluded

5.2.2 Study Objects

Following the previously explained methodology, the first 298 Java projects on the repository were analyzed for potential candidates for an integration of this action. From these 298, seven projects were left to perform the integration when applying the requirements.

Table 5.2: Java Project Statistics

Java Project	Lines of Code	Tests	Stars	Contributors
ReMap [73]	14070	189	108	6
Auto [74]	39958	422	10200	78
Jedis [75]	67906	825	11200	198
Guice [76]	73042	1468	12000	77
fastjson [77]	186876	5022	25400	179
ta4j [78]	27096	1141	1800	68
ZXing [79]	47273	564	31200	118

5.2.3 Experimental Integration

As a successful case of this integration, the Jedis [75] project was found. Jedis is the Java client for Redis [80], a popular in-memory database with over 11,000 stars and close to 200 contributors. For this project, it was possible to configure the execution of the developed action, producing a pull request with an example of execution in a previous commit with errors, as well as an introduction to the world of SBFL, GZoltar, and the action itself [5]. The pull request with the introduction can be seen in Figure 5.1, and it contains pointers to a sample run in the Jedis repository in a previous commit where the tests failed. The main report for this sample can be seen in Figure 5.2.

While the setup process was a success, the Jedis maintainers, despite finding the project fascinating, decided it was still too early to integrate it into their development process. Their response, shown in Figure 5.3, shows that there is interest in a solution of this type but that it may need more maturation and proof given to be integrated into a large-scale project.

When carrying out this task, unanticipated problems arose, making it challenging to obtain the intended results. Firstly, a difference emerged between the results of the tests with the execution without the GZoltar agent in relation to the execution with it. There was an example of this case when trying to integrate the fastjson tool that served to create an issue in the GZoltar repository for further investigation by the developers [81], taking that tool out of the equation.

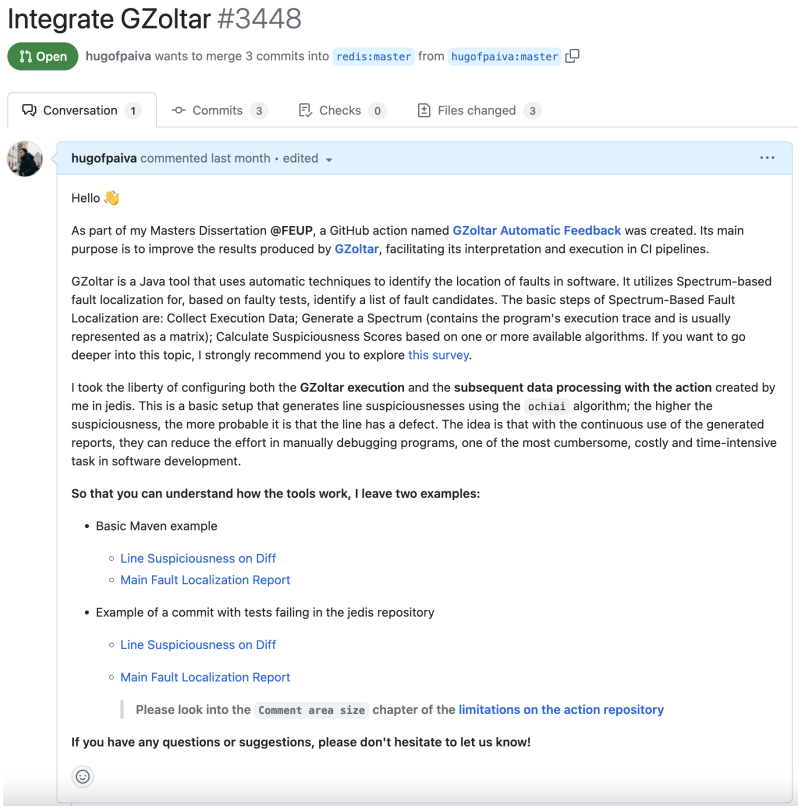


Figure 5.1: Pull Request with the integration of GZoltar and the action in the Jedis project [5]

Secondly, problems were constantly detected in projects that used the *Dependency Management* or *Plugin Management* to facilitate dependency management in multi-module projects when including GZoltar dependencies in the project during test execution. The most common error referred to JUnit's inability to find the GZoltar listener that would store and process the test execution information provided by JUnit. Therefore, all these projects could not be used since the execution of the tests was not possible, in addition to the high complexity that some of them had due to the architecture with multiple modules.

In order to summarize all these results, Table 5.3 was created. Speaking of numbers, one project out of seven managed to be integrated, representing a success rate of 14.28%.

Table 5.3: Summary of attempts to integrate the developed action into open-source projects

Java Project	**GZoltar Action Execution**		
	Success	**Tests Failing**	**Dependency/Plugin Management issues**
ReMap	No	No	Yes
Auto	No	No	Yes
Jedis	Yes	-	-
Guice	No	No	Yes
fastjson	No	Yes	No
ta4j	No	No	Yes
ZXing	No	No	Yes

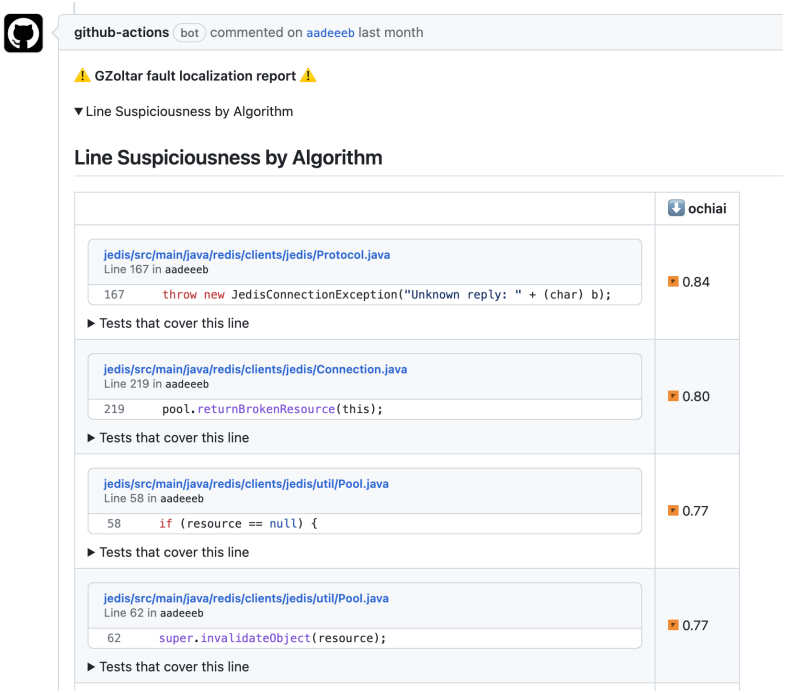


Figure 5.2: GZoltar report for the Jedis project in a previous commit where the tests failed [6]

Fundamentally, despite the less desired results, since the GZoltar agent is responsible for the collection of the information of the execution of the tests and the generation of the report, it is possible to realize that the major problem turns out to be related to GZoltar and its dependencies and not to the action itself.

5.2.4 Challenges

In addition to the limitations listed above, namely the discovery at this stage related to *Dependency Management* or *Plugin Management*, others were found that were tackled.

Node Memory Usage This action loads all the content resulting from the GZoltar processing into memory. This is necessary to quickly analyze possible faults and provide the maximum amount of information to the user; however, it turned out to be partly a scalability issue.

In the action's early iterations, the test case data model contained both lines that were covered and those that were not. When attempting to execute the activity on larger projects, this rapidly became a problem. To fix this, users were allowed to increase the amount of memory the node might consume while executing using the *NODE_OPTIONS* environment variable, and only the covered lines were preserved. In this manner, this issue was not discovered in any later tests.

Comment Area Size GitHub assigns a maximum width of 780px in the comments area. Although this may be sufficient for most cases, when there are long lines of code, the table can be unformatted, making it difficult to understand its content.

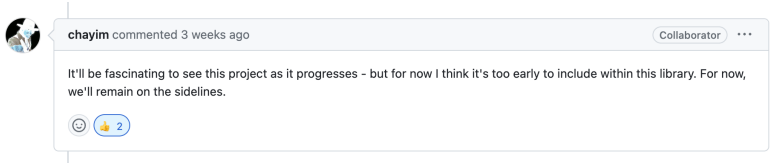


Figure 5.3: Response from the Jedis maintainers to the pull request with the integration of GZoltar GitHub Action in the Jedis project [5]

To solve this problem, a simple JavaScript script that needs to be executed in the browser console so that the maximum width size is increased was given as a solution. This script is present on the Listing 5.1, and the difference in visualization can be seen in Figure 5.4.

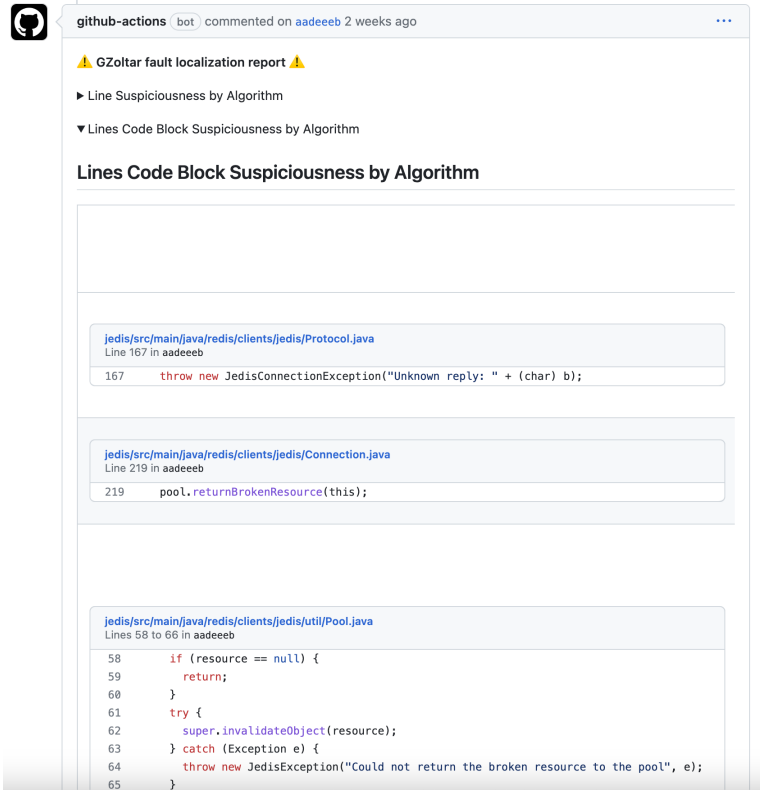
```
1    // Find the div element with id "comments"
2    var commentsDiv = document.getElementById("comments");
3    // Check if the div element exists
4    if (commentsDiv) {
5      // Set the maxWidth of the div element to 2000px
6      commentsDiv.style.maxWidth = "2000px";
7    }
```

Listing 5.1: JavaScript scrip to increase the maximum width size of the comments area to be run in a browser console [11]

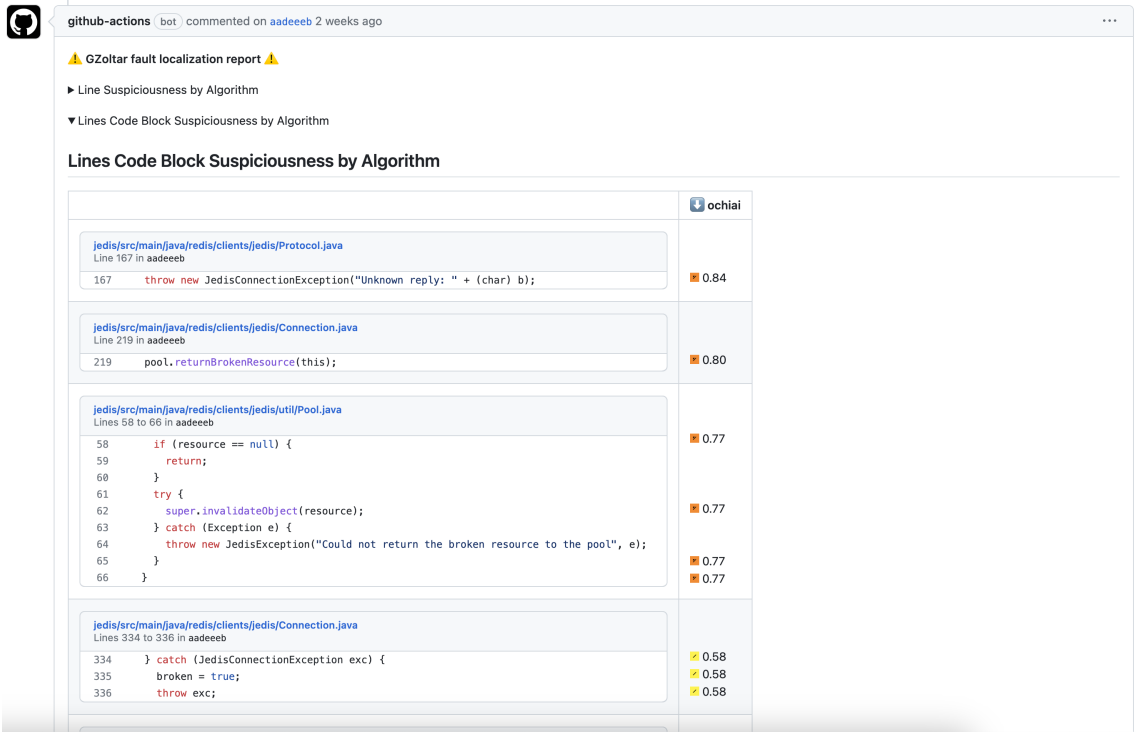
5.3 Summary

In this chapter, a qualitative evaluation between the developed action and the FLACOCOBOT was carried out, demonstrating several differences between the two.

However, the main focus was the presentation of the discussion of the results obtained with the execution of the action developed in open-source projects. A methodology was developed and followed, allowing to reduce the amount of open-source projects considered. The results obtained with the execution of the action in seven open-source projects were presented, as well as the challenges encountered during the execution of the action. Despite not having been able to integrate the action into the development process of any of these projects, several lessons were learned that can be used to improve the action and, thus, increase the probability of success of future integrations. All these lessons and conclusions will be discussed in the next chapter.



(a) Before the JavaScript script execution



(b) After the JavaScript script execution

Figure 5.4: Difference in the visualization of the comments area before and after the JavaScript script execution

Chapter 6

Conclusions and Future Work

6.1 Conclusion

The focus of this Dissertation was to create a simple and quick-to-setup tool capable of applying automated techniques in continuous integration environments to help with the software fault localization process. Of all software fault localization techniques existing, SBFL was chosen as the technique to be used in this work. The decision was made based on its low execution time and, also on the fact that it was used in 41% of research publications in the field of software fault localization.

With the emergence of several SBFL tools and, since the Java language is in the top-2 most used languages, an analysis of the existing tools in the Java ecosystem was carried out. It was concluded that among the 3 most popular tools (GZoltar, FLACOCO and Jaguar), GZoltar stood out due to the support of more SBFL out-of-the-box formulas, for being considered one of the most performing tools and for the involvement of supervisors of this work in its assignment.

Furthermore, an analysis beyond the most popular existing continuous integration tools was conducted in order to choose the one that would be used in this work. Among the choices, GitHub Action was chosen based on the fact that has a tight integration with GitHub, the most widely used code host service. This integration allows reaching a greater number of developers with a minimum of additional configuration, in addition to its generous free tier. Last but not least, this platform provides a marketplace where it is possible to find various CI tools created by the community, in addition to allowing the publication of new ones.

Therefore, a GitHub Action was created which, when integrated with GZoltar, allows the execution of SBFL in any Java repository on GitHub. The outcome is a detailed report of suspect lines of code according to user-defined settings.

The created action was tested in a publicly available example repository as well as in several open-source projects. While the integration with the sample repository was successful, the integration with open-source projects was partially successful as it was not possible to understand the impact and usefulness of the action in the development process. In addition, out of 298 analyzed Java projects, only 7 were amenable to integration, mainly due to the limitations of GZoltar.

Of these 7 projects, only one of them was successfully executed, again, due to GZoltar limitations. These limitations, previously discussed in Section 5.2, demonstrated that GZoltar needs to be updated in order to support the new version of JUnit, and also to be further tested in larger projects in order to increase its compatibility. In the only successful case, although maintainers found the project fascinating, they decided that it was still too embryonic to be integrated into their development process.

During this integration work, barriers were also overcome regarding the limitations of the GitHub platform as well as the memory management of the developed action.

Despite all these limitations and challenges, the action developed is successfully working and was published in the GitHub Marketplace, being available for use in any Java-based GitHub repository.

6.2 Future Work

6.2.1 GitHub Annotations

Instead of creating comments on the pull request or commit diff, it is possible to create annotations [82]. Annotations are similar to comments but can be attached to multiple lines or even specific parts of a line of code, pointing back to the action execution. Annotations can include a title, a message, and details. Furthermore, they have multiple levels of severity, representing either a notice, warning, or failure.

Annotations are visible on GitHub in the pull request's Checks and Files changed tab. To create them, the Checks API is used to update the status of the GitHub Action execution, where the annotations are included. This API limits the number of annotations to a maximum of 50 per API request, having to make multiple requests to update a check run endpoint in case more are needed. Each time a check run is updated, annotations are appended to the list of annotations that already exist for the check run. GitHub Actions are limited to ten warning annotations and ten error annotations per step.

Annotations are used in the previously mentioned tool *ESLint* when it is running in an action worker. Since this tool is a static code analyzer for identifying problematic patterns found in *JavaScript* code, it is natural that this feature is used to warn the detected problems. Figure 6.1 shows an example of these annotations where it is possible to see not only the information generated by the action that ran *ESLint* but also the association to the action that executed it, something that does not happen in a normal comment.

The use of annotations would have been an added value in this action as it would have facilitated the creation of information in the diff itself since it not only allows adding a report to lines present in the diff but those that are not present. Furthermore, it allows associating annotations with several lines instead of just one, as with comments.

Despite its advantages, this feature was not implemented as it is not widely discussed in the GitHub documentation. It only appears as a slight reference in the Check Runs API documentation

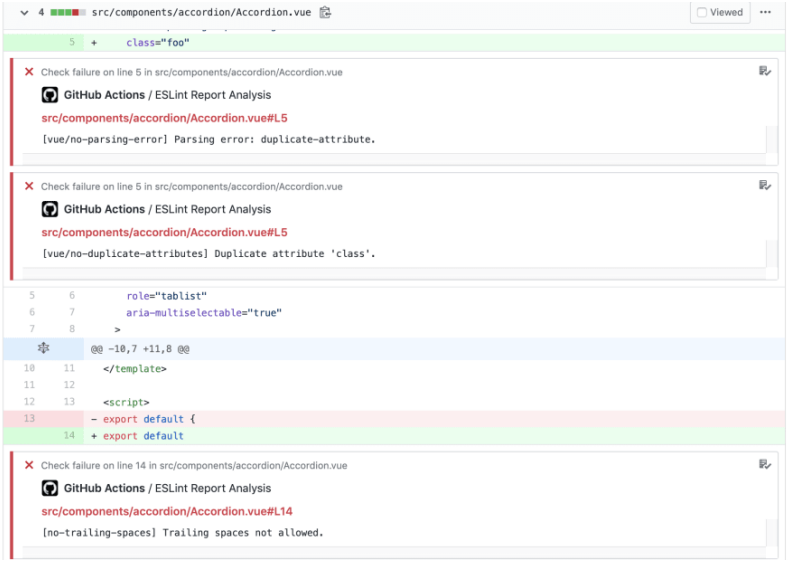


Figure 6.1: Example of *ESLint* annotations [7]

[82], which led to not being discovered initially during the development of this work. Still, it seems to be the most suitable GitHub functionality for this use case since it is associated with the action itself, instead of being a loose comment, in addition to the advantages mentioned above. It is still unknown whether this feature solves the limitation of the comment box size.

6.2.2 Browser Extension

Even though the GitHub platform is quite serviceable, there are limitations. An example of this was the size of the comments, which will undoubtedly make it challenging to interpret the results in projects with extensive lines of code since it will be necessary, at least, to run the script created for this purpose.

A browser extension capable of running GZoltar or just the developed action logic having the GZoltar report files would have allowed any user to have the GZoltar information on any GitHub repository, even without having writing permissions to the repository or even a GitHub account. This would allow users to include more information anywhere on the page, such as icons directly on lines of code. Furthermore, it might even be possible to cover more code platforms since as soon as the extension detects that the user is browsing a Java code repository, a suggestion for interpreting discovered GZoltar report files could be displayed.

6.2.3 Unit Tests

The unit tests aim to validate that each software unit performs as designed. Although examples of the action use have been created, and attempts have even been made to integrate the action into open-source projects, unit tests were not implemented. If there had been time available to do so, the testing framework to be used would have been the *Jest*, one of the most popular testing frameworks for *JavaScript* [83]. The main test point would have to be the parsing of the GZoltar report files,

ensuring that all the information generated by GZoltar was correctly stored, and the generation of tables with the suspiciousness information provided in the GitHub comments, ensuring that the information given to the user was also according to the one generated by GZoltar.

References

[1] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42:707–740, 8 2016.

[2] André Silva, Matias Martinez, Benjamin Danglot, Davide Ginelli, and Martin Monperrus. Flacoco: Fault localization for java based on industry-grade coverage. Technical Report 2111.12513, arXiv, 2021.

[3] Henrique Lemos Ribeiro, Higor Amario De Souza, Roberto Paulo Andrioli De Araujo, Marcos Lordello Chaim, and Fabio Kon. Jaguar: A spectrum-based fault localization tool for real-world software. *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, pages 404–409, 5 2018.

[4] Github actions. Available at https://docs.github.com/en/actions.

[5] Integrate gzoltar by hugofpaiva · pull request 3448 · redis/jedis. Available at https://github.com/redis/jedis/pull/3448.

[6] example: Gzoltar with fail test · hugofpaiva/jedis@aadeeeb. Available at https://github.com/hugofpaiva/jedis/commit/aadeeebf44ce3c8d2c285b1460e43ce2d04f0d2d#commitcomment-115720588.

[7] How to visualize eslint errors on github! - dev community. Available at https://dev.to/jnybgr/how-to-visualize-eslint-errors-on-github-3f8p.

[8] Archana and Ashutosh Agarwal. Evaluation of spectrum based fault localization tools. *ACM International Conference Proceeding Series*, 2 2022.

[9] Creating a javascript action - github docs. Available at https://docs.github.com/en/actions/creating-actions/creating-a-javascript-action.

[10] Simple example of gzoltar github action. Available at https://github.com/GZoltar/example-gzoltar-github-action.

[11] Run javascript in the console - chrome developers. Available at https://developer.chrome.com/docs/devtools/console/javascript/.

[12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, pages 378–381, 2012.

[13] Aaron Ang, Alexandre Perez, Arie Van Deursen, and Rui Abreu. Revisiting the practical use of automated software fault localization techniques. *Proceedings - 2017 IEEE 28th International Symposium on Software Reliability Engineering Workshops, ISSREW 2017*, pages 175–182, 11 2017.

[14] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 7 2016.

[15] Souza HAd, Lauretto MdS, Kon F, and Chaim M. Understanding the use of spectrum-based fault localization. 11 2022.

[16] Abubakar Zakari, Sai Peck Lee, Khubaib Amjad Alam, and Rodina Ahmad. Software fault localisation: a systematic mapping study. *IET Software*, 13:60–74, 2 2019.

[17] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. *Proceedings - 18th IEEE International Conference on Automated Software Engineering, ASE 2003*, pages 30–39, 2003.

[18] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. An evaluation of similarity coefficients for software fault localization. *Proceedings - 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006*, pages 39–46, 2006.

[19] Birgit Hofer, Alexandre Perez, Rui Abreu, and Franz Wotawa. On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering*, 22:47–74, 3 2015.

[20] Sivasurya Santhanam, Tobias Hecking, Andreas Schreiber, and Stefan Wagner. Bots in software engineering: a systematic mapping study. *PeerJ Computer Science*, 8:e866, 2 2022.

[21] Linda Erlenhov, Francisco Gomes de Oliveira Neto, and Philipp Leitner. An empirical study of bots in software development – characteristics and challenges from a practitioner's perspective. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 20:445–455, 5 2020.

[22] Rahul Kumar, Chetan Bansal, Chandra Maddila, Nitin Sharma, Shawn Martelock, and Ravi Bhargava. Building sankie: An ai platform for devops. *Proceedings - 2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering, BotSE 2019*, pages 48–53, 5 2019.

[23] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. What to expect from code review bots on github?: A survey with oss maintainers. *ACM International Conference Proceeding Series*, pages 457–462, 10 2020.

[24] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? *Proceedings - 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021*, pages 420–431, 3 2021.

[25] Continuous integration tools market size, share, growth - 2023 | marketsandmarkets. Available at https://www.marketsandmarkets.com/Market-Reports/continuous-integration-tools-market-154327001.html.

[26] Jenkins. Available at https://www.jenkins.io/.

[27] Travis ci. Available at https://www.travis-ci.com/.

[28] Circleci. Available at https://circleci.com/.

[29] Gitlab ci/cd | gitlab. Available at https://docs.gitlab.com/ee/ci/.

[30] The state of developer ecosystem in 2021 infographic | jetbrains: Developer tools for professionals and teams. Available at https://www.jetbrains.com/lp/devecosystem-2021/.

[31] Visual studio code - code editing. redefined. Available at https://code.visualstudio.com/.

[32] Apache ant - welcome. Available at https://ant.apache.org/.

[33] Maven - welcome to apache maven. Available at https://maven.apache.org/.

[34] Javassist by jboss-javassist. Available at https://www.javassist.org/.

[35] Matias Martinez and Martin Monperrus. Astor: A program repair library for java (demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, 7 2016.

[36] Carlos Gouveia, José Campos, and Rui Abreu. Using html5 visualizations in software fault localization. *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.

[37] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, Martin Monperrus, and Sebastian Lamelas Marcote. Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43:34–55, 2017.

[38] Jacoco java code coverage library. Available at https://www.jacoco.org/jacoco/.

[39] Junit 5. Available at https://junit.org/junit5/.

[40] GitHub. Network dependents · jacoco/jacoco. Available at https://github.com/jacoco/jacoco/network/dependents?package_id=UGFja2FnZS0xNzk5NDIyMDk%3D.

[41] Github - rjust/defects4j: A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research. Available at https://github.com/rjust/defects4j.

[42] Eclipse ide | the eclipse foundation. Available at https://eclipseide.org/.

[43] Marcos Lordello Chaim and Roberto Paulo Andrioli De Araujo. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, 113:293–300, 4 2013.

[44] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault localization in concurrent programs. *Proceedings - International Conference on Software Engineering*, 1:245–254, 2010.

[45] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. Vida: Visual interactive debugging. *Proceedings - International Conference on Software Engineering*, pages 583–586, 2009.

[46] About billing for github actions - github docs. Available at https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions.

[47] Github marketplace · actions to improve your workflow. Available at https://github.com/marketplace?type=actions.

[48] Github marketplace welcomes its 10,000th action | the github blog. Available at https://github.blog/2021-10-21-github-marketplace-welcomes-its-10000th-action/.

[49] The global developer community | the state of the octoverse. Available at https://octoverse.github.com/2022/developer-community.

[50] Enterprise open source and linux | ubuntu. Available at https://ubuntu.com/.

[51] actions/checkout: Action for checking out a repo. Available at https://github.com/actions/checkout.

[52] actions/setup-node: Set up your github actions workflow with a specific version of node.js. Available at https://github.com/actions/setup-node.

[53] bats-core/bats-core: Bash automated testing system. Available at https://github.com/bats-core/bats-core.

[54] About custom actions - github docs. Available at https://docs.github.com/en/actions/creating-actions/about-custom-actions.

[55] actions/toolkit: The github toolkit for developing github actions. Available at https://github.com/actions/toolkit.

[56] Gzoltar · actions · github marketplace. Available at https://github.com/marketplace/actions/gzoltar.

[57] Find and fix problems in your javascript code - eslint - pluggable javascript linter. Available at https://eslint.org/.

[58] Prettier · opinionated code formatter. Available at https://prettier.io/.

[59] Typescript: Javascript with syntax for types. Available at https://www.typescriptlang.org/.

[60] vercel/ncc: Compile a node.js project into a single file. supports typescript, binary addons, dynamic requires. Available at https://github.com/vercel/ncc.

[61] Encrypted secrets - github docs. Available at https://docs.github.com/en/actions/security-guides/encrypted-secrets.

[62] Quickstart for writing on github - github docs. Available at https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/quickstart-for-writing-on-github.

[63] Coloradd - color is for all! Available at https://www.coloradd.net/.

[64] Git - viewing the commit history. Available at https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History.

[65] Junit support update by blackalbino17 · pull request 68 · gzoltar/gzoltar. Available at https://github.com/GZoltar/gzoltar/pull/68.

[66] Maven repository: junit » junit. Available at https://mvnrepository.com/artifact/junit/junit.

[67] Cannot create issue - comment is too long (maximum is 65536 characters) · community · discussion 41331. Available at https://github.com/orgs/community/discussions/41331.

[68] Gzoltar maven example. Available at https://github.com/GZoltar/gzoltar/tree/master/com.gzoltar.maven.examples.

[69] Linux statistics 2023 | 99firms. Available at https://99firms.com/blog/linux-statistics/.

[70] Db-engines ranking - popularity ranking of database management systems. Available at https://db-engines.com/en/ranking.

[71] akullpp/awesome-java: A curated list of awesome frameworks, libraries and software for the java programming language. Available at https://github.com/akullpp/awesome-java.

[72] Gradle build tool. Available at https://gradle.org/.

[73] remondis-it/remap: A declarative mapping library to simplify testable object mappings. Available at https://github.com/remondis-it/remap.

[74] google/auto: A collection of source code generators for java. Available at https://github.com/google/auto.

[75] redis/jedis: Redis java client. Available at https://github.com/redis/jedis.

[76] google/guice: Guice (pronounced 'juice') is a lightweight dependency injection framework for java 8 and above, brought to you by google. Available at https://github.com/google/guice.

[77] alibaba/fastjson: Fastjson 2.0.x has been released, faster and more secure, recommend you upgrade. Available at https://github.com/alibaba/fastjson.

[78] ta4j/ta4j: A java library for technical analysis. Available at https://github.com/ta4j/ta4j.

[79] zxing/zxing: Zxing ("zebra crossing") barcode scanning library for java, android. Available at https://github.com/zxing/zxing.

[80] Redis. Available at https://redis.io/.

[81] When introducing gzoltar to a pipeline of tests, some of the tests start to fail · issue 70 · gzoltar/gzoltar. Available at https://github.com/GZoltar/gzoltar/issues/70.

[82] Check runs - github docs. Available at https://docs.github.com/en/rest/checks/runs?apiVersion=2022-11-28#create-a-check-run.

[83] Jest · delightful javascript testing. Available at https://jestjs.io/.