On the Investigation of Exception Pull Request Characteristics: Exploring the pache Ecosystem

João Correia , Daniel Coutinho , lessandro Garcia , Rafael de Mello[†], Caio Barbosa , nderson Oliveira ,
Wesley K. G. ssunção[‡], Juliana lves Pereira , Igor Steinmacher[¶], Marco Gerosa[¶], Jairo Souza[§], Johny rriel
Pontifical Catholic University of Rio de Janeiro - Brazil, [†]*Federal University of Rio de Janeiro - Brazil*,
[‡]*North Carolina State University - US* , [§]*Northern rizona University - US* , [¶]*Federal University of Pernambuco - Brazil*
jcorreia, dcoutinho, afgarcia}@inf.puc-rio.br, rafaelmello@ic.ufrj.br, csilva, aoliveira}@inf.puc-rio.br, wguezas@ncsu.edu,
juliana@inf.puc-rio.br, igor.steinmacher, marco.gerosa}@nau.edu, jrmcs@cin.ufpe.br, johny.arriel@les.inf.puc-rio.br

***bstract*—Robustness is critical for ensuring that software functions correctly under adverse conditions. Exception-handling mechanisms in programming languages enable developers to deal with these adverse conditions. However, implementing exception-related code can present significant challenges to developers. We investigated exception-related code contributions across Java projects in the pache ecosystem. We analyzed exception-related pull requests (exception-PRs), which were detected using a validated heuristic. We produced a comprehensive dataset of 988 exception-PRs. We observed no statistically significant differences in complexity metrics between exception-PRs and non-exception-PRs. We also found no significant differences in developers' behavior metrics, indicating consistent engagement regardless of whether the pull request addressed exception-related code. manual analysis revealed that most exception-PRs focused on system improvements rather than bug fixes, suggesting proactive efforts to enhance software robustness. Moreover, the most frequently addressed aspects of exceptional code in these exception-PRs were: (i) the external representation of adverse situations to end-users (more than 40% of the PRs) and (ii) the implementation of effective error-handling actions (nearly 35% of the PRs) to promote program recoverability. Interestingly, a significant proportion of exception-PRs simultaneously addressed multiple aspects. By understanding the nature and characteristics of exception-PRs, we expect to better support developers in managing erroneous conditions and improving software robustness.**

***Index Terms*—software robustness, exception handling, exceptions, maintenance, evolution**

I. Introduction

Software robustness is of paramount importance to the success of a system. Robustness is described as the "degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [1]. Thus, the essential part of a program to achieve software robustness is the one that manages such erroneous inputs or conditions [2]–[6]. Certain programming languages offer a specific mechanism, the so-called *exception mechanism* or *exception handling mechanism* [5], [7]–[10], to deal with these erroneous conditions and structure this part of the program.

Java is an example of a programming language with a built-in exception-handling mechanism. Java enables developers to define `try`, `catch`, and `finally` blocks as well as raise or propagate exceptions with `throw` and `throws` statements [11].[1] With these language features, developers can ensure the system's recoverability (*i.e.*, catch or handle exceptions) from invalid or stressful conditions [7], [12]. Developers can also represent these conditions as *exceptions*, as well as propagate them to upper layers. They can also implement the system's cleanup actions through `finally` blocks [10], [13]. The part of the program that manages (*i.e.*, raise, protect, clean up, and handle) exceptions can be called the *exception-related code*.

Over the decades, various studies have investigated the inherent challenges in properly implementing exception-related code [14]–[16]. In the early 80s, Black *et al.* [17] warned developers about the complexity of exception-related code in object-oriented programs. In the late 2000s, Shah *et al.* [2] examined the quality of exception-related code in certain industrial software projects, concluding that, despite its importance to program robustness, its quality might be overlooked by developers. Further analysis of the same authors suggests that using exception mechanisms is hard for less experienced developers [3]. Given these reasons, Shah *et al.* [2] even proposed the introduction of a new role of *exception engineer* to industrial software projects; this role would be dedicated to the design, implementation, and maintenance of the exception behavior of a software system.

In the context of open-source systems, one might question the nature of the contributions to the exception-related code and how they differ from other types of contributions. However, no empirical study has characterized and classified contributions to the exception-related code in open-source systems. Previous studies have focused on analyzing the frequency of changes to different sections of the exception-related code [9] and how these changes relate to bugs [6], [18]. While some studies [4], [19], [20] have investigated the nature of exception-related bugs, other studies (*e.g.*, [21]–[25]) focused on providing automatic repair for such bugs.

This paper reports a study we performed aimed at understanding the nature of contributions to the exception-related code, or *exception-related pull requests* (*exception-PRs*, for short), in Java projects of the pache ecosystem. Our study also aimed to analyze the attention devoted by developers to exception-PRs as compared to other types of contributions in these projects. To achieve our aims, we implemented a heuristic to detect exception-PRs that, according to our validation, reached a precision of 90.7% (254) of the 280 manually evaluated PRs. Our study included several metrics,

[1]https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

such as number of reviews, the number of requests for reviews, the number of comments. Our statistical analysis to compare the dynamics of exception-PRs with non-exception-PRs is inconclusive in determining differences in developers working on these two categories of PRs.

We also manually analyzed the information in these 254 exception-PRs to understand the types of contributions made by developers. The results of this analysis reveal the multifaceted aspects of these contributions. The principal aim for opening an exception-PR is improvement, accounting for 55.12% (140 out of 254) of the PRs. Bug fix is the second most frequent aim, representing 40.55% (103 out of 254) of the PRs. The most frequently modified aspect of exception-PRs involves the external error representation, appearing at least once in approximately 42.52% (108 out of 254) of the PRs. This means that the primary focus of these PRs is enhancing the clarity of error messages for users. Effective error handling is the second most frequent focus of the PRs, occurring in 36.22% (92 out of 254). Effective error handling concerns the proper implementation of exception handlers (catch blocks) to ensure the system's recoverability. lthough addressed less frequently, contributions related to error detection and propagation were still notable, appearing in 23.62% (60 out of 254) and 16.54% (42 out of 254) of the PRs. Notably, when looking at these aspects of the exception-related code, external error representation often appears alone in about 18.50% of cases and roughly twice as frequently as part of combined modifications in 35.04% of PRs. Similarly, error detection and propagation are more often involved in combined contributions.

II. REL TED WORK

Shah *et al.* [2] conducted an empirical study to characterize exception-handling practices in industrial projects. The authors interviewed developers to understand their perspective on exception handling and their strategies for handling exception constructs. The results revealed developers' dissatisfaction with exception mechanisms in programming languages (such as Java) mainly due to the language imposition to implement such constructs. lso, developers claimed to have limited support in implementing exception-handling constructs. Based on these results, the authors propose the new role of *exception engineer*, who should focus on designing, implementing, and maintaining the exception-related code in software systems.

In a following study, Shah *et al.* [3] interviewed developers from the industry to characterize the problems and obstacles they face when designing and implementing exception handling. The results revealed a trend that novice developers usually ignore exceptions and frequently use them primarily to support debugging activities. In contrast, experts consider both regular and exception-related functionalities similarly important, using exception-handling mechanisms to convey important information to users.

Nakshatri *et al.* [26] conducted an empirical study to validate the findings from Shah *et al.* [2]. The authors examined common practices for exception handling in Java projects, comparing them to the best practices outlined in the book Effective Java [27]. Furthermore, they analyzed the comparison results alongside the evidence provided by Shah *et al.* [2]. The results revealed that most developers often overlook checked exceptions and tend to use higher-level exception classes more frequently than lower-level ones.

Sawadpong *et al.* [28] conducted an exploratory study investigating the defect density in exception handling and the overall source code. The authors explored the code of six releases in the Eclipse IDE and computed the defect ratios. Their results suggested that the exception defect density is roughly three times the overall code defect density. This observation reinforces previous works' findings that exception handling is usually neglected.

Ebert *et al.* [4], [29] conducted an exploratory study on exception-handling bugs. They classified 220 bugs from Eclipse and Tomcat repositories in terms of frequency, severity, and difficulty in fixing them. Next, they surveyed 154 developers to understand organizational policies and perceptions about exception handling bugs. s a result, the authors proposed a categorization for exception-handling bugs and their causes. lso, they found in their analysis that exception-handling bugs are less ignored than other bugs. ccording to the survey, only 27% of the respondents claimed that their organization has practices for implementing exception-handling mechanisms.

Viviani *et al.* [30], [31] investigated pull request discussions through manual classification of three open-source systems to find design-related topics. The results revealed that the three most frequent topics were *maintainability*, *code*, and *robustness*. Moreover, while the studies above suggest that exception-related code is often overlooked in coding activities, robustness remains a crucial concern of design discussions. These findings present an intriguing opportunity for further exploration, which could lead to a more comprehensive understanding of the challenges faced by implementing exception-related code in software systems.

Our present study differs from the existing literature in various ways. First, we examine the content of exception-related pull requests to understand the types of contributions to exception-related code in open-source systems. Second, we study how the dynamics of exception-related and non-exception-related contributions differ. Third, we depict and classify the aspects of exception-related code contributions.

III. STUDY DESIGN

This section describes the research questions and the methodology we employed to answer them. We gathered data from three pache systems. Subsequently, we categorized the PRs into exception-PRs and non-exception-PRs and conducted quantitative and qualitative analyses, as described below.

. *Research Questions*

We aim to analyze the characteristics of contributions concerning the exception-related code in open-source projects. To this end, we compared PRs related to exceptions (exception-PRs) with the other PRs (non-exception-PRs). dditionally, we examined the dynamics among developers within these contributions to identify the presence of exception experts. Finally, we also examined the exception aspects commonly addressed in exception-PRs. Our study is structured around the following research questions:

RQ_1: *How does the dynamics of exception-PRs and non-exception-PRs differ?* To answer this research question, we compare exception-PRs metrics with non-exception-PRs ones. We hypothesize that exception-PRs are more complex than non-exception-PRs due to some characteristics. Firstly, exception-related code often addresses issues to ensure the resilience and stability of software systems under diverse operational conditions. Thus, a deeper understanding of the system's architecture and the potential fault lines that could compromise its integrity is required. Furthermore, exception-related code usually requires rigorous testing activities to cover a broader range of scenarios, including edge and uncommon cases. Concluding that exception-PRs are more complex may imply that they require more attention from team managers regarding resource allocation and process optimization.

In this *RQ*, we also aim to understand whether developers' behavior changes between exception-PRs and non-exception-PRs. For instance, we want to understand whether developers put more effort into exception-PRs than non-exception-PRs and identify developers whose team often relies on performing exception-related code changes. This analysis is grounded in results from previous work [30] that reveal the experts' participation in discussions related to robustness. lso, Shah *et al.* [2] suggests that developers working on the system's exception-related code should have specialized knowledge.

RQ_2: *What are the aspects addressed by exception-PRs?* Given that robustness requirements are often retrofitted during development, and previous research has strongly emphasized bug-fixing of exception-related code, this research question is driven by the need to understand why developers submit exception-related code contributions. We seek to systematically classify the aims of exception-PRs, determining whether they introduce new functionalities, enhance existing ones, or address defects. dditionally, we aim to comprehend the aspects of exception-related code frequently addressed by such contributions, focusing on error representability, error recoverability, error detection, error propagation, and their subcategories. By answering this research question, we aim to enrich our understanding of how contributions manage exception-related code issues and whether existing tools designed to aid in writing exception-related code effectively support developers. To our knowledge, no previous research has undertaken efforts to understand the characteristics of exception-related code contributions in this manner.

B. Data Collection

We focused our studies on a set of projects that satisfied the following criteria:

Open-source systems. s aforementioned, the primary focus of our investigation lies on open-source systems, which also favors data availability and the future replication of this research. Open-source systems are typically maintained through version control systems, which provide unrestricted access to historical data. This access would be difficult to have in closed-source systems. We selected projects with repositories in GitHub since the platform has an application programming interface (PI) for retrieving repository data, including pull requests, issues, comments, users, and commits. We rely on this data to answer the aforementioned RQs.

Java language. This study focuses on Java projects, as the language provides mechanisms that push for robustness guarantees [6], [9], [18]. Java also has an exception-handling mechanism, allowing developers to determine exception-related conditions and explicitly handle them. This mechanism helps developers build resilient applications that recover from exception-related situations without crashing. s the language provides explicit exception-handling features, we can more objectively analyze how systems deal with robustness requirements. Java is a statically typed language, meaning that type-checking is performed at compile-time. Many exceptions are checked for having at least a handler defined at compile time. This language characteristic may help catch various errors early in development, reducing the likelihood of runtime errors and enhancing the program's robustness.

pache ecosystem. The pache ecosystem comprises a collection of open-source systems developed and maintained by a diverse community. Due to factors such as the relevance of systems, the number of contributors, and the need for robust non-functional requirements, several studies have chosen the pache ecosystem to empirically investigate reliability [32]–[34]. In this study, we selected three projects from this ecosystem considering the following criteria: (i) the repository should have more than 5,000 commits; (ii) the repository contains commits pushed a month before our data collection; (iii) the repository should have recent discussions on PRs and issues; and (iv) the repositories are from different domains.

Under the criteria above, we selected the following repositories: pache Druid, pache Pulsar, and pache Flink. We systematically mined and collected data from these repositories, covering their entire history until July 2023. By choosing repositories from different domains, we aspire to foster heterogeneity in our study.

1) Heuristic: fter defining the repositories, we designed a heuristic to filter contributions related to exceptions. This heuristic is aimed to be conservative regarding labeling contributions as exception-related in an attempt to enhance precision. Therefore, the heuristic comprised three conditions:

$_1$ The title must contain the keyword *exception* or *throw* or *catch* or its derivatives [2]; and

$_2$ Body must contain the keyword *exception* or *throw* or *catch* or its derivatives; and

$_3$ The PR must not have been opened by a *bot*.

We employed the heuristic approach for each PR within the repository to determine its association with exception-related

[2]One of the considered words with the addition of a prefix or a suffix.

code. Initially, the heuristic analyzed the title and description of the PR to check its adherence to the criteria $_1$ and $_2$. Subsequently, the criterion $_3$ was checked by leveraging a feature provided by the GitHub PI v3. Thus, the heuristic identified PRs initiated by bots and disregarded them in an attempt to remove noise from our dataset. Table I provides an overview of the collected data. The st column shows the repository name. The 2^{nd} column shows the amount and percentage of exception-PRs in the repository. The 3^{rd} column shows the total amount of PRs in the repository. Finally, the 4^{th} column describes the domain of the system. s the data implies, we obtained 988 or 2.12% of the PRs classified as exception-PRs. In related work, Ebert *et al.* [4] obtained a similarly low number when analyzing only exception-handling bugs from Bugzilla in the Eclipse and Tomcat repositories.

T BLE I
REPOSITORIES ND DET ILS BOUT D T COLLECTION.

Repository	exception-PRs	Total PRs	Domain
druid	126 (1.30%)	9675	Real-time analytics database
pulsar	294 (2.07%)	14198	Messaging/streaming platform
flink	568 (2.49%)	22728	Distributed processing engine

2) Metrics: To support the analyses to answer RQ_1, we collected a set of metrics from pull requests and the developers. Table II presents these metrics and their descriptions. For each pull request, we computed the *number of reviews*, the *number of requests to review*, and the *number of comments*. For each developer, we calculated the *percentage of pull requests*, *percentage of reviews*, *percentage of requests to review*, and *percentage of comments*. Before computing these metrics, we split the PRs and developers according to their relation to exceptions using our heuristic. For instance, we calculated the developers' metrics for exception-PRs and non-exception-PRs, instead of computing their metrics in all PRs. In other words, the developer's metric *reviews ratio* has one value for exception-PRs and another for non-exception-PRs. In this way, we can perform statistical metrics analyses for both scenarios.

T BLE II
PULL REQUESTS' ND DEVELOPERS' METRICS DESCRIPTION.

	Metric	Description
Pull request	Number of reviews	Number of reviews made by contributors on that pull request.
	Number of comments	Number of comments made by contributors on that pull request.
	Number of requests to review	Number of contributors marked as reviewers on that pull request.
Developer	Pull requests ratio	Percentage of pull requests opened by that user in the sample.
	Reviews ratio	Percentage of reviews made by that user in the sample.
	Requests to review ratio	Percentage of requests to review for that user in the sample.
	Comments ratio	Percentage of comments made by that user in the sample.

3) Manual validation: We conducted a manual validation process with experienced validators, focusing on two main objectives. The first was to evaluate the precision of our heuristics in identifying real exception-PRs, thereby ensuring confidence in the analysis conducted for RQ_1. The second, addressing RQ_2, was to characterize the aim of exception-PRs and exception-related code aspects impacted by contributions.

We initially calculated the sample size of exception-PRs needed for manual validation, ensuring it would represent our entire population of 988 with 95% confidence. The estimated sample size was 277 PRs. Therefore, we randomly selected 280 exception-PRs from the dataset, distributed as follows: 160 from Flink, 82 from Pulsar, and 38 from Druid.

We requested participants to collaborate in the validation process by sending an invitation message. The message contained the invitation and instructions for participating. Eight developers joined the manual validation process, each evaluating 70 PRs. Table III provides the demographics of these developers in terms of self-rated experience with software development, software exceptions, and pull request collaborations. Notice that we managed the validation process so that we had a double validation for each of the 280 PRs.

T BLE III
V LID TOR'S DEMOGR PHICS.

Validator	Experience with Software Development	Experience with Software Exceptions	Experience with Pull Requests
V1	Moderate	Moderate	Very high
V2	Very high	High	Very low
V3	High	Moderate	Moderate
V4	High	Low	High
V5	Very high	High	Very high
V6	Moderate	Moderate	Low
V7	High	Moderate	Moderate
V8	High	Very high	High

Below, we present the three questions that validators answered for each PR. We have the complete description of options in our supplementary material [35].

Q_1 *What was the aim of the pull request?* Single choice question, with the following alternatives:

- **New feature:** addition of a completely new feature to the system.
- **Improvement:** extension or refactoring of existing code structures.
- **Bug fix:** correction of a malfunction.
- **Test:** addition/change in code for test.
- **Other:** other non-covered aim.

Q_2 *Is this pull request related to an exception?* Single choice question, with the following alternatives:

- **Yes:** the pull request is related to exception.
- **No:** the pull request is not related to exception.

Q_3 *Which exception-related aspect(s) was touched by the code contribution?* Multiple choice question, with the following alternatives:

- **Error representability:** whether errors are properly represented/specified.

  Internal: whether errors are properly represented in the software.

  External: whether information about error detection, propagation, or handling is properly specified, traced, or logged for further consideration by the developer or user.

– **Error recoverability:** whether proper exception handling and clean-up actions are executed after an exception is raised, and the program's normal behavior returns to a consistent/safe state after those actions are executed.

  Effective error handling: whether the proper handlers and their actions are statically or dynamically attached to exceptions.

  Program continuity: whether the continuation of the program after error recovery is returned to the proper place in the program or the program abruptly stopped.

  Cleanability: whether the clean-up actions are properly chosen and activated in the program through the exception flow.

– **Error detectability:** whether errors are properly detected/thrown, *i.e.*, whether errors are raised and with the proper raising conditions.

– **Error propagation:** whether the error is properly propagated by either remapping to another error type or directly propagated to upper levels.

– **Other (non-exception-related code):** whether the error was solved without touching any of the aforementioned attributes.

s we performed a double validation process, disagreements were expected. To resolve disagreements on questions Q_1 and Q_2, four authors collaboratively reviewed and reconciled differences. For Q_3, we consolidated responses by merging the responses of both validators, since we understand that one code contribution could address complementary aspects perceived by different validators.

C. Data nalysis

This section describes the methods for analyzing the collected data and answering the RQs. We explain the steps performed in our data analysis as follows.

Step 1. s illustrated in Table II, we gathered a set of metrics for each repository in this study. We then analyzed these metrics in two subsets: exception-PRs and non-exception-PRs. Both sets were created using the heuristic described in Section III-B1. We were expecting in our initial hypothesis that exception-PRs would demonstrate greater complexity than non-exception-PRs. By complexity, we refer to a high degree of interaction between developers and the code review process. Consequently, we conducted hypothesis tests to evaluate each PR metric (Table II) to address RQ_1. Below, we present the null and alternative hypotheses considered in our tests:

> The PR's metric in exception-PRs is not greater than non-exception-PRs.
> The PR's metric in exception-PRs is greater than non-exception-PRs.

We conducted the *Mann–Whitney test* [36] for each metric of PRs. In this scenario, the null hypothesis (H) suggests that the PR' metric in exception-PRs is not greater than that in non-exception-PRs. In contrast, the alternative hypothesis (H) indicates that the PR' metric in exception-PRs is greater than in non-exception-PRs.

In addition to the hypothesis test, we applied *Cliff's delta* analysis [37] to obtain valuable insights into the magnitude of the differences observed between the PR metrics in exception-PRs and non-exception-PRs. While the Mann-Whitney test would establish a statistically significant difference between the two groups of metrics, *Cliff's delta* complements this analysis by quantifying the effect size, thereby offering a more comprehensive understanding of the practical significance of the findings.

Step 2. In the previous step, we examined the complexity of exception-PRs and non-exception-PRs. Here, we aim to investigate whether there is a difference in developers' behavior when dealing with exception and non-exception PRs. s mentioned in Section III-B2, we computed the developer's metrics for both subsets of exception-PRs and non-exception-PRs. We rely on these metrics (Table II) to observe the developers' behavior. To compare the resulting measures, we conducted a statistical test based on the following hypotheses:

> The developer's metric in exception-PRs is not greater than non-exception-PRs.
> The developer's metric in exception-PRs is greater than in non-exception-PRs.

Since we expected to compare the mean values of variables in the same population, we employed the paired *Wilcoxon test* [38] for statistical analysis. This test is well-suited for analyzing matched pairs of data. dditionally, we utilized *Cliff's Delta* to assess the magnitude of the observed results.

IV. Results nd Discussions

. Heuristic Precision

s outlined in Section III-B3, a manual validation process was conducted to assess the heuristic precision in detecting exception-PRs. The validators agreed that 90.7% (254 out of 280) of the PRs analyzed were related to exception, in contrast to 9.3% (26 out of 280) that were not. These results confirm the heuristic's high precision in identifying exception-PRs, bolstering confidence in our analyses for RQ_1.

We gathered further insights by analyzing cases where validators disagreed on their relationship to exceptions. The analysis provided valuable information on the limitations of the heuristic and helped us refine our understanding of what constitutes an exception-related contribution. Notice PR #17078[3] from Flink, which involved a change in test procedures. Specifically, it threw the original exception rather than failing with a JUnit assertion if OpenSSL was not found. This change, impacting only test files and not system behavior, underscores the heuristic's challenge in distinguishing between changes that affect system exception-related code and those that do not. nother example, PR #14207[4] from Pulsar, involved unnecessary state transition handling in the TransactionCoordinatorClientImpl.start sync method. The decision to remove the

[3]https://github.com/apache/flink/pull/17078
[4]https://github.com/apache/pulsar/pull/14207

redundant exception also does not affect the system behavior, illustrating the nuanced nature of contributions that impact the system's exception-related code. In summary, most of the non-exception-PRs classified by validators, cited exceptions. However, these contributions did not change the system's functional behavior; therefore, the validators considered them unrelated to exception management.

B. Categorizing the ims of Exception-PRs

During the manual validation, we requested participants to classify the developers' aim for opening PRs. We obtained the following results by examining only the exception-PRs (254). The principal aim for opening an exception-PR was *improvement*, accounting for 55.12% (140) of the PRs. *Bug fix* was the second most frequent aim, representing 40.55% (103) of the PRs. *Test* was the aim of 3.15% (8) of the PRs, followed by *new feature* as the fourth most frequent aim, with 0.79% (2) of the cases. Finally, only 0.39% (1) PR was attributed to *other* miscellaneous reasons.

This result was interesting from the perspective of the existing literature on mining exception changes. Even though most of the previous studies focus on investigating the nature of exception bugs and their fixes (e.g., [4], [20], [28], [29]), such fixes were only the second most frequent aim in the analyzed PRs. In the projects analyzed, developers more frequently perform a wide range of exception-related improvements (than bug fixes), which we will discuss in more detail in Section IV-E. These improvements may vary from enhancing preventive or clean-up actions to altering error messages. This leads to our first finding.

Finding 1: While most empirical studies in the literature focus on characterizing and classifying exception bugs (e.g., [4], [20], [28], [29]), there is a limited understanding about the most recurring forms of exception-related improvements made in open source projects.

 nother surprising result was the low occurrence of exception-PRs aimed at creating new features. Our rationale is that since Java forces developers to handle exceptions as soon as functional features are added, new features related to robustness (i.e., creating a new exception type) come up with regular features. Thus, since the PR is meant to introduce the new functional feature, the robustness aspect of the contribution can be left aside in the PR's title and description. Then, when our heuristic looks for the exception-PRs, contributions aimed at new features are less frequent.

Finding 2: New features concerning the exception-related code are often introduced as a secondary or tertiary aim; they are often intertwined with the introduction of functional features.

C. The Dynamics of Exception-PRs and Non-exception-PRs

In this section, we aim to present the answers for RQ_1. With our heuristic proven reliable for detecting exception-PRs, we selected two datasets of respective exception-PRs and non-exception-PRs. We investigated PR's metrics and aimed to find significant differences, supporting the hypothesis that exception-PRs are more complex than non-exception-PRs. This complexity was quantified using metrics like the *number of reviews*, the *number of requests to review*, and the *number of comments* on PRs. We hypothesized that exception-PRs would involve more detailed discussions and feedback, reflected in higher values for these metrics. The results for the hypothesis tests are presented in Table IV.

T BLE IV
HYPOTHESIS TEST FOR PR'S METRICS

	Metric	p-value	Cliff's Delta	Outcome
Druid	Number of reviews	0.412	negligible	Fail to reject .
	Number of requests to review	0.074	negligible	Fail to reject .
	Number of comments	0.033	negligible	rejected.
Flink	Number of reviews	0.999	negligible	Fail to reject .
	Number of requests to review	0.071	negligible	Fail to reject .
	Number of comments	0.810	negligible	Fail to reject .
Pulsar	Number of reviews	0.565	negligible	Fail to reject .
	Number of requests to review	0.354	negligible	Fail to reject .
	Number of comments	0.937	negligible	Fail to reject .

The statistical analysis fails to reject the H across the three repositories for most metrics, except for the *number of comments* for Druid. In this repository, the number of comments on exception-PRs showed a statistically significant difference compared to non-exception-PRs, with a *p-value* of 0.033, indicating rejection of the H . However, the effect size was negligible as measured by *Cliff's Delta*.

For the other two repositories, Flink and Pulsar, the hypothesis tests for all the metrics did not result in rejecting the H . The *p-values* were substantially higher than the conventional alpha level of 0.05, which would be required to declare a statistically significant difference. The effect sizes were negligible, suggesting that any differences between the exception-PRs and non-exception-PRs for these metrics are of minimal practical significance. Such results imply that while exceptions are an essential aspect of software development, we cannot reject H in favor of H , which suggests more complexity in exception-PRs. This leads to our third finding.

Finding 3: Contributions to exception-related code are so intertwined with the implementation of regular features that naturally lead to both receiving a similar amount of review comments and catching the attention of the same developers. This finding does not seem to back up the need for the role of "exception engineering" (Section II), as a previous study [2] suggested.

D. Developers in Exception-PRs and Non-exception-PRs

The earlier analysis indicates that the lack of difference is apparent between the composition of exception-PRs and non-exception-PRs when considering PR metrics. Looking further, we also explored developer metrics across these two categories of PRs. We still wanted to understand whether an "exception engineering" occurs in these projects. To this end, we calculated individual metrics for developers and applied a paired statistical hypothesis testing method. This approach helped identify significant differences in developers' activities when working on exception-PRs versus non-exception-PRs. Table V presents the results for the hypothesis tests; in the following paragraphs, we discuss them in detail.

T BLE V
HYPOTHESIS TEST FOR DEVELOPER'S METRICS

	Metric	p-value	Outcome
Druid	Pull requests ratio	0.008	rejected.
	Reviews ratio	0.846	Fail to reject .
	Requests to review ratio	0.965	Fail to reject .
	Comments ratio	0.353	Fail to reject .
Flink	Pull requests ratio	0.160	Fail to reject .
	Reviews ratio	0.990	Fail to reject .
	Requests to review ratio	0.430	Fail to reject .
	Comments ratio	0.990	Fail to reject .
Pulsar	Pull requests ratio	0.006	rejected.
	Reviews ratio	0.645	Fail to reject .
	Requests to review ratio	0.429	Fail to reject .
	Comments ratio	0.611	Fail to reject .

For both systems Druid and Pulsar, the H that there is no difference in the developers' *pull requests ratio* between exception-PRs and non-exception-PRs was rejected (p-values of 0.008 and 0.006, respectively). Initially, it suggests that developers in these projects are significantly more likely to open exception-PRs than non-exception-PRs. This result could indicate a specialization or higher focus on exception-related issues within these projects. However, for Flink, the *p-value* of 0.160 suggests no significant difference in the engagement level in exception-PRs compared to other non-exception-PRs.

cross all three projects, the *p-values* for *reviews ratio, requests to review ratio, and comments ratio* were high, leading to a failure to reject the H . This indicates that the number of reviews conducted, requests to review made, or comments posted between exception-PRs and non-exception-PRs is not different. This consistent lack of significant differences across these metrics suggests that despite there being developers that might be more likely to initiate exception-PRs in specific projects, once these PRs are created, they do not attract more reviews or comments than other PRs, reinforcing the observations in the Section IV-D.

The significant findings in the *pull requests ratio* for Druid and Pulsar, but not for Flink, could indicate varying cultural or procedural norms in these projects or differences in the complexity or impact of exception-related code issues identified in these contexts. It suggests that projects having developers with a higher propensity to submit exception-PRs might benefit from specialized training or resources to enhance the quality and effectiveness of these contributions, thereby improving the overall exception management of the software.

In summary, our analysis for RQ_1 reveals that we were unable to reject the H in favor of the H , indicating no differences in interaction levels, such as *number of reviews*, *requests for reviews*, and *number of comments* between exception-PRs and non-exception-PRs, except a slight increase in comments within the Druid repository. Interestingly, despite this statistical significance, the *Cliff's Delta* for this metric was negligible, suggesting that the practical impact of these differences is minimal. In contrast, developer metrics performed notable differences in the *pull request ratios* for Druid and Pulsar, suggesting a propensity for developers to initiate exception-PRs in these projects.

Finding 4: lthough exception-PRs do not generally attract more engagement than non-exception-PRs, the results indicate that specific projects may culturally or procedurally highlight the importance of triggering exception-PRs.

E. Exception spects in Contributions

Previous studies focus only on characterizing exception bugs (Section II). In the following sections, we present results related to RQ_2, where we observed a much wider variety of exception contributions. This research question aims to determine which aspects exception-PRs frequently address. In this section, we present frequencies for the following exception aspects: *Error representability (internal or external), Error recoverability (effective error handling, program continuity or cleanability), Error detection,* and *Error propagation*. In the next section IV-F, we analyze the PRs that address multiple of these aspects combined as part of a single contribution.

During our manual validation, we categorized each exception-PR according to up to three exception aspects. fter that validation, we calculated the prevalence of exception aspects across the 254 exception-PRs. Table VI illustrates our results. The st column lists each exception aspect, 2^{nd} and 3^{rd} respectively present the percentage and number of PRs where that aspect shows up. The subsequent columns detail the aim of the PRs having that aspect.

1) Error representability: is a critical aspect of software robustness, ensuring that errors are accurately and effectively communicated both within the system (internal representation) and to its users (external representation), see Section III-B3. Ideally, each project should have its own guidelines for crafting internal and external error messages, though not all do, or they may rely on very generic guidelines. Writing effective error messages that adhere to these guidelines can be challenging. Such representations must consider the semantic context of where the error was raised and propagated, as well as align with changes in the overall system.

External error representability: This exception aspect was the most frequently modified, accounting for 42.52% (108 PRs). The primary aim for these changes was overwhelmingly for program improvements, representing 77.77% of the cases. Bug fixes constituted 19.44%, while new feature implementations and tests were relatively minor. The high percentage of improvements suggests a significant focus on enhancing the clarity and usefulness of error messages for end-users. This leads us to another interesting finding.

Internal error representability: This aspect was the fourth most modified, accounting to 20.47% (52 PRs). Improvements were again the dominant aim, making up 63.46%, followed by bug fixes at 36.53%. There were no new features or tests associated with these changes. The focus on improvements here indicates efforts to refine how exceptions are represented internally within the system, which is important for better supporting program debugging and maintenance.

T BLE VI
EXCEPTION SPECTS ND THE IM OF THE CONTRIBUTION IN EXCEPTION-PRS.

Exception aspect occurrence			Pull Request im				
spect	**%**	**#**	**Bug fix**	**Improvement**	**New feature**	**Test**	**Other**
External error representability	42.52%	108	19.44%	77.77%	1.85%	0.92%	0.00%
Error recoverability - Effective error handling	36.22%	92	48.91%	46.73%	2.17%	1.08%	1.08%
Error detectability	23.62%	60	40.00%	56.67%	0.00%	3.33%	0.00%
Internal error representability	20.47%	52	36.53%	63.46%	0.00%	0.00%	0.00%
Error recoverability - Program continuity	20.47%	52	55.76%	42.30%	0.00%	1.92%	0.00%
Other (non-exception-related code)	19.69%	50	68.00%	26.00%	0.00%	6.00%	0.00%
Error propagation	16.54%	42	26.19%	69.05%	2.38%	2.38%	0.00%
Error recoverability - Cleanability	7.48%	19	36.84%	63.15%	0.00%	0.00%	0.00%

Finding 5: Most automated solutions in the literature (e.g., [22], [23], [25], [39]–[41]) focus on assisting developers in improving or repairing catch-block handlers. However, most improvements in exception-PRs aim to enhance the usability and usefulness of error messages.

2) Error recoverability: is crucial for maintaining system stability and reliability in the face of errors. It is usual for developers to postpone the implementation of comprehensive error-handling practices until the latter stages of development, prioritizing the integration of new features. lso, in the latter stages, the developer has more insights into error patterns. Effective *Error recoverability* involves error handling practices and considerations for program continuity and cleanability (see Section III-B3). Below, we discuss the frequency of each sub-aspect of recoverability.

Effective error handling: Overall, this was the second most frequently touched aspect, appearing in 36.22% (92 PRs). The distribution of aims was balanced, with 48.91% for bug fixes and 46.73% for improvements. New features and tests were minimal. The significant proportion of bug fixes may indicate a reactive approach to addressing error-handling issues as they arise, while the substantial number of improvements points to ongoing efforts to enhance error recovery mechanisms.

Program Continuity: Touched by 20.47% (52 PRs), this attribute saw 55.76% of changes aimed at bug fixes and 42.30% at improvements. The high frequency of bug fixes highlights the need to ensure the system continue operating smoothly even when errors occur, minimizing user disruptions.

Cleanability: This attribute was the less touched, appearing in only 7.48% (19 PRs). The contributions to aspects aimed at improvements (63.15%) and bug fixes (36.84%). The focus on improvements suggests efforts to enhance the system's ability to recover from errors and clean up appropriately, preventing error accumulation and potential system degradation over time.

In summary, effective error handling, the second most addressed exception aspect, shows a notable focus on bug fixes, maybe indicating a reactive approach to resolving issues that disrupt the correct system functioning. Program continuity and cleanability, though less modified, reflect efforts to maintain uninterrupted system functionality and improve error recovery mechanisms. These findings underscore the system's proactive behavior in mitigating disruptions caused by errors and enhancing overall reliability.

3) Error detectability: ensures that errors are identified promptly and accurately; see the Section III-B3. This aspect was the third most modified accounting 23.62% (60 PRs). Improvements were the principal aim (56.67%), followed by bug fixes (40.00%) and tests (3.33%). The higher numbers of improvements may indicate an effort to enhance the system's ability to catch errors early since issues with *Error detectability* may result in failures raising to end-user. Conversely, the percentage of bug fixes may suggest challenges in managing error detection in early development phases.

4) Error propagation: concerns the management of error propagation throughout the system (refer to Section III-B3. This aspect received less attention, accounting for 16.54% (42 PRs). The changes aimed at improvements constituted 69.05%, with bug fixes at 26.19%, and both new features and tests at 2.38%. The significant focus on improvements may suggest a delayed attention to error propagation. One rationale is that errors are initially only handled locally (where they occur), and as the system evolves, they are propagated and handled in the proper layers. Since error propagation is primarily an architectural concern, the low percentage of bug fixes may be due to this aspect not directly impacting the end-user experience as significantly as other issues do.

5) Other (non-exception-related code): not all exception-related contributions necessarily modify the exception-related aspects of the code. For example, a `FileNotFoundException` might be thrown during a file read operation, and the fix could involve inserting an if block to check for the file's existence before attempting to read it. For this and other scenarios, we allowed validators to classify some exception-PRs touching other aspects. *Other* was the sixth most modified aspect, accounting for 19.69% (50 PRs) of the cases. Unlike the other aspects discussed, *Other* was related to bug fixes 68.00% of the time, improvements 26.00% of the time, and tests 6.00% of the time.

Our analysis reveals a stronger emphasis on improvements across all exception aspects, indicating a proactive approach to enhancing the system's exceptions. Bug fixes also constitute significant changes, underscoring the ongoing need to address existing issues. The relatively low focus on new features may also suggest that, for Java, exception management is more about refining and solidifying existing capabilities rather than introducing new ones. While present, tests are less frequent since validators classified most PRs as only handling test files

T BLE VII
COMBIN TION OF EXCEPTION SPECTS ND THE IM OF CONTRIBUTION IN EXCEPTION-PRS

Exception aspects occurrence (combinations)			**Pull Request im**				
spect	**%**	**#**	**Bug fix**	**Improvement**	**New feature**	**Test**	**Other**
Error recoverability, Error representability	12.99%	33	39.39%	57.58%	3.03%	0.00%	0.00%
Error detectability, Error recoverability, Error representability	7.09%	18	44.44%	55.56%	0.00%	0.00%	0.00%
Error propagation, Error recoverability, Error representability	5.51%	14	42.86%	50.00%	7.14%	0.00%	0.00%

unrelated to exceptions.

F. Multiple Exception spects Touched by Contributions

In the previous section, we examined the frequency exception aspects disregarding combinations. In this section, we focus on analyzing the combinations of exception aspects. By exploring how these elements co-occur within exception-PRs, we aim to gain deeper insights into the integrated strategies employed to enhance system exceptions. The Table VII illustrates the top three most frequent combinations in exception-PRs. The complete list of combinations is available in our supplementary material [35]. The st column lists each exception aspect combination, 2^{nd} and 3^{rd} respectively present the percentage and number of exception-PRs where that aspect occurred. The following sections discuss the top three combinations with real examples.

1) Error recoverability and Error representability: was the most common combination found in our manual validation, constituting 12.99% (33 PRs). This combination mainly characterizes contributions that bond how errors are classified, handled, and reported. The principal aim of contributions with this combination were improvements in 57.58% and bug fixes in 39.39%, revealing a slight tendency for improvement in these aspects rather than bug fix issues. For instance, we identified examples on *improving the understandability of the exception handling*. In one of these cases[5], a developer stated that "It's hard to investigate [issue number]. dding the thread dump might help us to identify the cause, or at least having a pointer to what to focus on". In this case, the developer was investigating an issue and found that the existing error reporting mechanisms were insufficient, so he made changes to improve that. We also observed cases where developers encountered the opposite situation (i.e., excessive error reporting). n example of this was a situation described by a developer in a PR #1107[6]: "This [situation described in the issue] can print several thousand exception stack traces." Since he found this behavior excessive, he proposed a solution to reduce the number of exceptions logged.

nother prevalent scenario that we observed was where the existing class modeling an exception was not satisfactory (*e.g.,* because it does not adequately describe the error, there is a more fitting exception that already exists elsewhere in the code[7], etc.). While fixing a bug[8], one developer described this as adding a "meaningful exception" to the error handling. We also observed cases[9] where the existing exceptions in the code were insufficient to express (and report) a specific error, so a new exception class was introduced.

2) Error detectability, Error recoverability, and Error representability: was the second most frequent in our analysis, representing 7.09% (18 PRs), and its distribution in terms of aim is similar to the previous combination, improvements 55.56% and bug fixes 44.44%. In our observation, this combination mainly emerged for two reasons: (i) disagreement between validators and (ii) large and complex PRs. While these reasons can be observed separately, they blend into each other, as most disagreements emerged from PRs that were not easy for the validators to understand. This is understandable, given that while the validators had extensive language experience, they may not be experts in the project.

PRs exhibiting these aspects of the combination were typically very complex, with varied characteristics. However, most shared the common trait of affecting multiple parts of the exception pipeline. For example, consider a PR #2264[10] submitted to the Druid project. When an exception occurred, the system would simply return a generic error to the user. To address this, the developer proposed a significant change by introducing a module called `ustomExceptionMapper`, designed to handle specific exception types. This module generates a detailed error message tailored to the particular exception and uses it to provide the user with more informative feedback. dditionally, the module was designed to be generic, enabling it to handle other exception types similarly. This change exemplifies the pattern by addressing three key aspects: detectability, recoverability, and representability.

3) Error propagation, Error recoverability, Error representability: was the third most frequent combination, mirroring the characteristics described in the previous pattern. This specific pattern accounts for 5.51% (14 PRs) of the cases, with its goals distributed as follows: 50.00% aimed at improvement and 42.86% at bug fixes. We observe that contributions to that aspect also address complex issues, primarily on *Error propagation*. Propagation is often a challenging topic for developers [42], as it requires in-depth knowledge of how exceptions are handled at an architectural level. This handling is often far from trivial in large systems like those we analyzed, a fact noted by validators. This complexity also makes validating these types of PRs more difficult, especially since the validators were not project experts. Consequently, PRs adhering to this pattern often propose extensive and

[5]https://github.com/apache/flink/pull/21463
[6]https://github.com/apache/pulsar/pull/1107
[7]https://github.com/apache/flink/pull/4701
[8]https://github.com/apache/flink/pull/2969
[9]https://github.com/apache/flink/pull/8509
[10]https://github.com/apache/druid/pull/2264/

intricate changes, such as altering handlers so that exceptions are managed at a different architectural level (e.g., client-side instead of server-side[11]), or fixing exception handling within asynchronous environments[12].

G. Single versus Combined Exception spects

Table VIII illustrates that exception aspects frequently occur in combination rather than isolation. For example, *Error representability* appears in combination in 35.04% of PRs, compared to only 18.50% focusing exclusively on this aspect. Similarly, *Error recoverability* is observed in 33.46% of PRs in combined scenarios, contrasting with 12.20% as a single aspect. This trend suggests developers often address multiple exception-related aspects simultaneously when suggesting exception-related contributions.

T BLE VIII
SPECTS OF EXCEPTION-PRS PPE RING SINGLE OR IN COMBIN TION.

	Single	Combination
Error representability	18.50%	35.04%
Error recoverability	12.20%	33.46%
Error detectability	5.51%	18.11%
Error propagation	2.76%	12.20%
Other	17.32%	2.35%

The prevalence of combined exception aspects may indicate two points about exception-related contributions. First, it underscores the interconnection of exception aspects. For example, improving error handling may concurrently require adjustments in *Error representability* to maintain message consistency. Second, it highlights the complexity of modern software systems, where exception management is not achieved through isolated contributions but through integrated strategies. Interestingly, we observe an inverse pattern for the *other* category, which happened more separately than combined. This result suggests that contributions not touching the exception-related code occur in isolation without impacting other exception aspects.

Finding 6: Developers frequently address multiple exception aspects simultaneously in exception-PRs, emphasizing the interrelation and complexity of exception management in modern software systems.

V. THRE TS TO V LIDITY

Internal Validity. Our study analyzed three projects from pache. While this limited scope might suggest a potential threat to the generality of our findings, the selected projects provided a substantial amount of data from different domains, allowing for robust analysis within a specific community. Regarding the heuristic, which was designed to select exception-PRs, it may not capture all pertinent changes. However, our heuristic was specifically designed to capture changes focused on exception, aligning closely with our research objectives. dditionally, by concentrating on human code contributions, we prioritized the quality and relevance of the data, ensuring its significance for our study.

[11]https://github.com/apache/pulsar/pull/7430
[12]https://github.com/apache/flink/pull/9115

External Validity. The results of this study are based on a carefully selected set of projects. In this matter, this selection process ensures depth and relevance in the studied context, even while limiting the applicability of our findings to similar environments or communities. We conducted manual validations by specialized validators using a sized sample. This may induce a subjective interpretation, potentially affecting the replicability of the validation process. However, the process included a double validation, where four authors resolved conflicts collaboratively. This approach minimizes human error and bias, ensuring high reliability of the data validation.

VI. CONCLUSION

In this study, we explored the nature of exception-PRs in Java projects within the pache ecosystem to understand the contributions to exception-related code and the attention they receive from developers compared to other types of contributions. Using a heuristic, we collected 988 exception-PRs from three repositories and selected 280 for manual analysis. We confirmed a precision of 90.7%, identifying 254 accurate exception-PRs out of the 280 evaluated. Our statistical analysis found no significant differences in developer engagement metrics, such as reviews and comments, between exception-PRs and non-exception-PRs. detailed manual analysis of the 254 exception-PRs revealed that 55.12% aimed at improvements, 40.55% focused on bug fixes, with the most frequently modified aspect being external error representation (42.52%), followed by effective error handling (36.22%).

These results suggest that, while exception handling is recognized for its critical role in software robustness, it does not necessarily command more or less attention from developers in the review process compared to other types of contributions. This finding raises important questions about the prioritization and attention of exception-related code, requiring further investigation. Moreover, by understanding the nature and characteristics of exception-PRs, we can better support developers in managing erroneous conditions and improving software robustness. Future work could explore automated tools and techniques to assist developers in creating and maintaining high-quality, exception-related code, ultimately contributing to more reliable and robust software systems.

CKNOWLEDGMENT

This work was supported by following grants. C PES: 175956, 88887.900069/2023-00, 88881.879016/2023-01, 88887.373933/2019-00, and 88887.899310/2023-00; CNPq: 404027/2023-7, 140770/2021-6, 315711/2020-5, 141180/2021-8, 141276/2020-7 and 141054/2019-0; F PESP: 2023/00811-0; F PERJ: 200.510/2023, 211.033/2019, 202.621/2019, and F PERJ PDR-10 program 202073/2020; NSF: 2236198, 2247929, 2303042, and 2303612; and IEE -RJ001/2021.

REFERENCES

[1] International Organization for Standardization, “ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary,” *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017.

[2] H. Shah, C. Görg, and M. J. Harrold, “Why do developers neglect exception handling?” in *4th International Workshop on Exception Handling*. CM, 2008, p. 62–68.

[3] H. Shah, C. Gorg, and M. J. Harrold, “Understanding exception handling: Viewpoints of novices and experts,” *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 150–161, 2010.

[4] F. Ebert, F. Castor, and . Serebrenik, “ n exploratory study on exception handling bugs in java programs,” *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[5] . F. Garcia, C. M. Rubira, . Romanovsky, and J. Xu, “ comparative study of exception handling mechanisms for building dependable object-oriented software,” *Journal of Systems and Software*, vol. 59, no. 2, pp. 197–222, 2001.

[6] N. Cacho, T. César, T. Filipe, E. Soares, . Cassio, R. Souza, I. Garcia, E. . Barbosa, and . Garcia, “Trading robustness for maintainability: an empirical study of evolving c# programs,” in *36th International Conference on Software Engineering*. CM, 2014, p. 584–595.

[7] P. . Buhr and W. R. Mok, “ dvanced exception handling mechanisms,” *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 820–836, 2000.

[8] M. P. Robillard and G. C. Murphy, “Static analysis to support the evolution of exception structure in object-oriented systems,” *CM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 2, pp. 191–221, 2003.

[9] B. Jakobus, E. . Barbosa, . Garcia, and C. J. Pereira de Lucena, “Contrasting exception handling code across languages: n experience report involving 50 open source projects,” in *26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 183–193.

[10] B. Jakobus, E. . Barbosa, . Garcia, and C. J. P. De Lucena, “Contrasting exception handling code across languages: n experience report involving 50 open source projects,” in *26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 183–193.

[11] M. saduzzaman, M. hasanuzzaman, C. K. Roy, and K. . Schneider, “How developers use exception handling in java?” in *13th International Conference on Mining Software Repositories*, 2016, pp. 516–519.

[12] L. Chung, B. . Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Springer Science & Business Media, 2012, vol. 5.

[13] C. Hagen and G. lonso, “Exception handling in workflow management systems,” *IEEE Transactions on software engineering*, vol. 26, no. 10, pp. 943–958, 2000.

[14] J. Petke, D. Clark, and W. B. Langdon, “Software robustness: a survey, a theory, and prospects,” in *29th CM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1475–1478.

[15] N. Laranjeiro, J. gnelo, and J. Bernardino, “ systematic review on software robustness assessment,” *CM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–65, 2021.

[16] . Shahrokni and R. Feldt, “ systematic review of software robustness,” *Information and Software Technology*, vol. 55, no. 1, pp. 1–17, 2013.

[17] . P. Black, “Exception handling: The case against,” Ph.D. dissertation, University of Oxford, 1982.

[18] N. Cacho, E. . Barbosa, J. raujo, F. Pranto, . Garcia, T. Cesar, E. Soares, . Cassio, T. Filipe, and I. Garcia, “How does exception handling behavior evolve? an exploratory study in java and c# applications,” in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 31–40.

[19] H. Zhong and H. Mei, “Mining repair model for exception-related bug,” *Journal of Systems and Software*, vol. 141, pp. 16–31, 2018.

[20] E. . Barbosa, . F. Garcia, and S. D. J. Barbosa, “Categorizing faults in exception handling: study of open source projects,” in *2014 Brazilian Symposium on Software Engineering*. IEEE Computer Society, 2014, pp. 11–20.

[21] B. Cabral and P. Marques, “ transactional model for automatic exception handling,” *Comput. Lang. Syst. Struct.*, vol. 37, no. 1, pp. 43–61, 2011.

[22] E. . Barbosa, . Garcia, and M. Mezini, “Heuristic strategies for recommendation of exception handling code,” in *26th Brazilian Symposium on Software Engineering*. IEEE Computer Society, 2012, pp. 171–180.

[23] E. . Barbosa, . F. Garcia, and M. Mezini, “ recommendation system for exception handling code,” in *5th International Workshop on Exception Handling*. IEEE, 2012, pp. 52–54.

[24] B. Cornu, L. Seinturier, and M. Monperrus, “Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions,” *Inf. Softw. Technol.*, vol. 57, pp. 66–76, 2015.

[25] E. . Barbosa and . Garcia, “Global-aware recommendations for repairing violations in exception handling,” *IEEE Trans. Software Eng.*, vol. 44, no. 9, pp. 855–873, 2018.

[26] S. Nakshatri, M. Hegde, and S. Thandra, “ nalysis of exception handling patterns in java projects: an empirical study,” in *13th International Conference on Mining Software Repositories*. CM, 2016, p. 500–503.

[27] J. Bloch, *Effective java*. ddison-Wesley Professional, 2008.

[28] P. Sawadpong, E. B. llen, and B. J. Williams, “Exception handling defects: n empirical study,” in *14th International Symposium on High-ssurance Systems Engineering*, 2012, pp. 90–97.

[29] F. Ebert, F. Castor, and . Serebrenik, “ reflection on “an exploratory study on exception handling bugs in java programs”,” in *2020 IEEE 27th International Conference on Software nalysis, Evolution and Reengineering (S NER)*, 2020, pp. 552–556.

[30] G. Viviani, C. Janik-Jones, M. Famelis, X. Xia, and G. C. Murphy, “What design topics do developers discuss?” in *26th Conference on Program Comprehension*. CM, 2018, p. 328–331.

[31] G. Viviani, M. Famelis, X. Xia, C. Janik-Jones, and G. C. Murphy, “Locating latent design information in developer discussions: study on pull requests,” *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1402–1413, 2021.

[32] . Yan, H. Zhong, D. Song, and L. Jia, “How do programmers fix bugs as workarounds? an empirical study on apache projects,” *Empirical Software Engineering*, vol. 28, no. 4, p. 96, 2023.

[33] H. Chen, W. Dou, Y. Jiang, and F. Qin, “Understanding exception-related bugs in large-scale cloud systems,” in *2019 34th IEEE/ CM International Conference on utomated Software Engineering (SE)*, 2019, pp. 339–351.

[34] V. Piantadosi, S. Scalabrino, and R. Oliveto, “Fixing of security vulnerabilities in open source projects: case study of apache http server and apache tomcat,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 68–78.

[35] nonymous, “Replication package,” 2024, accessed on: 2024-06-23. [Online]. vailable: https://github.com/opus-research/robustness-prs-replication

[36] Y. Dodge, *The concise encyclopedia of statistics: Mann–Whitney Test*. Springer Science & Business Media, 2008, pp. 327–329.

[37] C. Zaiontz, “Cliff’s delta,” 2024, accessed: 2024-06-18. [Online]. vailable: https://real-statistics.com/non-parametric-tests/mann-whitney-test/cliffs-delta/

[38] D. Rey and M. Neuhäuser, *Wilcoxon-Signed-Rank Test*. Springer Berlin Heidelberg, 2011, pp. 1658–1659.

[39] E. . Barbosa and . Garcia, “Global-aware recommendations for repairing violations in exception handling,” in *40th International Conference on Software Engineering*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. CM, 2018, p. 858.

[40] E. . Barbosa, . F. Garcia, M. P. Robillard, and B. Jakobus, “Enforcing exception handling policies with a domain-specific language,” *IEEE Trans. Software Eng.*, vol. 42, no. 6, pp. 559–584, 2016.

[41] L. S. Rocha, R. M. C. ndrade, and . F. Garcia, “ method for model checking context-aware exception handling,” in *27th Brazilian Symposium on Software Engineering*. IEEE Computer Society, 2013, pp. 59–68.

[42] X. Qiu, L. Zhang, and X. Lian, “Static analysis for java exception propagation structure,” in *2010 IEEE International Conference on Progress in Informatics and Computing*, vol. 2, 2010, pp. 1040–1046.