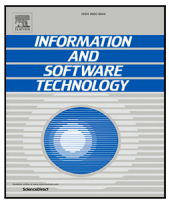
Information and Software Technology 199 (2026) 108264

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Exception handling bugs in Python: An empirical study of root causes, fix patterns, and anti-patterns

Jairo Souza [a,*], Eric Coelho [b], João Correia [c], Rodrigo Lima [a], Leopoldo Teixeira [a], Baldoino Fonseca [b]

[a] Federal University of Pernambuco (UFPE), Informatics Center, Recife, Pernambuco, Brazil
[b] Federal University of Alagoas (UFAL), Maceió, Alagoas, Brazil
[c] Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

ARTICLE INFO

Keywords:
Exception handling
Empirical study
Bugs
Anti-patterns

ABSTRACT

Context: Exception handling mechanisms are designed to manage abnormal events that disrupt normal program execution, helping software recover from unexpected situations. However, defects in exception handling mechanisms can introduce errors, crashes, and unexpected behavior, reducing system reliability. Despite widespread adoption of Python, exception handling bugs in Python projects remain largely underexplored.
Objectives: This study aims to investigate the characteristics of exception handling bugs in Python projects by analyzing their root causes, fix patterns, and their relationship with anti-patterns (exception handling anti-patterns).
Methods: We conducted a large-scale empirical study on 550 open-source Python projects. Using our Exception Miner tool and manual validation, we analyzed 942 confirmed exception handling bugs identified from 1,649 bug candidates extracted from issue reports and associated commits.
Results: Our analysis identified 12 root causes and 25 fix patterns. The most common root cause is *Unhandled Exception*, accounting for 50.64% of the analyzed cases. The most frequent repair strategies include adding exception handling blocks, changing exception types, and introducing appropriate `raise` conditions, which together account for approximately 60% of the fixes. A before-and-after analysis shows that many exception handling anti-patterns were preserved from the buggy version, while fixes still introduced an increase of 196 exception handling anti-patterns occurrences.
Conclusion: The results reveal systematic relationships between root causes and repair strategies, indicating that exception handling bugs often follow predictable patterns. These findings provide insights to improve testing practices and design automated tools to detect and repair exception handling bugs in Python projects.

1. Introduction

Exception handling is a fundamental aspect of programming, involving mechanisms for handling errors or exceptional events that occur during program execution and potentially disrupt the program's normal flow [1]. These exception handling mechanisms [2,3] (e.g., exception handling commands) are widely implemented across various programming languages such as Java, C#, and Python [4]. In Python, these mechanisms (herein called "exception handling mechanisms") are implemented through exception handling commands, such as `try`, `except`, `finally`, and `raise`, which enable the transition from normal to exception-based control flow. Python, in particular, is a popular programming language for open-source projects.[1] Exception handling mechanisms provide advantages to the software quality, such as increasing program readability, reliability, and maintainability [5].

Although exception handling mechanisms offer various benefits to software quality, their misuse can result in bugs [6] (hereafter called "exception handling bugs"). Beyond causing crashes, exception handling bugs can also lead to incorrect behavior, misleading diagnostics, and silent failures, reinforcing the need for a systematic investigation of their causes and fixes. Previous studies have investigated the relationship of exception handling mechanisms with aspects such as evolution [7,8], practices [9], adopted strategies [10,11], and robustness [12]. Other studies have focused on understanding exception handling bugs [6,13–15]. However, most of these studies analyze Java

* Corresponding author.
E-mail address: jrmcs@cin.upfe.br (J. Souza).
[1] https://www.tiobe.com/tiobe-index

https://doi.org/10.1016/j.infsof.2026.108264
Received 7 March 2026; Received in revised form 15 July 2026; Accepted 16 July 2026
Available online 22 July 2026

and C# projects, leaving open questions about how such exception handling bugs manifest in Python. Python differs from these languages in several ways, including its dynamic typing, lack of compile-time checks, and runtime dependency management [16], which could potentially lead to exception handling bugs with distinct root causes that may differ from those observed in previous studies.

In this paper, we conduct an exploratory study to understand exception handling bugs in Python projects by identifying and analyzing their root causes and fix patterns. We also investigate the impact of these fixes by analyzing whether their application contains anti-patterns associated with exception handling mechanisms (hereafter called "exception handling anti-patterns"). To this end, we collect 942 exception handling bugs from 550 open-source Python projects. Based on the exception handling bugs collected, our results reveal a taxonomy of 12 root causes and 25 fixes emerged from exception handling bugs, indicating the variety of exception handling bugs. This taxonomy provides a structured approach to categorizing root causes and fix patterns into distinct groups. We also summarize the main fix patterns for each exception handling bug and analyze how exception handling anti-patterns evolve before and after bug fixing. The results show that many anti-patterns were already present in the buggy version, but the fixes still led to an increase of 196 anti-pattern occurrences. While these categories offer a comprehensive view of how Python developers handle exceptions, many of them align with findings from studies on Java and C# developers [6,10,13,15]. This suggests that, despite Python's distinctive language features, exception handling bugs often share common characteristics across languages.

In summary, the main contributions of our study are:

- We perform the first comprehensive study to understand exception handling bugs in Python;
- We provide a taxonomy of 12 distinct root causes categories for the exception handling bugs collected in our study;
- We summarize the common fix patterns related to exception handling bugs;
- We provide a dataset containing 942 exception handling bugs existing in Python projects;

Our study and dataset[2] pave the way for researchers to develop new tools for detecting and fixing exception handling bugs and advance in the understanding by exploring new analyses on the relation between root causes, fix patterns, and exception handling anti-patterns. For developers, the list of root causes and fix patterns provides a valuable resource for more effective bug triaging and fixing.

2. Background

Exception handling is a fundamental programming language mechanism for handling abnormal situations that occur during program execution. According to the ISO/IEEE standard, exception handling enables the propagation of error information by throwing and catching exceptions [17], where an exception is defined as an event that suspends the normal program execution and indicates that an operation request was not performed successfully [17]. Its primary purpose is to ensure the robustness and reliability of software systems by preventing abrupt program termination and enabling programs to respond to or recover from unexpected situations [4,18].

Modern programming languages provide constructs to support exception handling. In Python, these mechanisms include constructs such as `try`, `except`, `finally`, and `raise`, which enable developers to detect exceptional conditions, recover from errors, and perform cleanup. These mechanisms improve software reliability and maintainability by separating normal-execution logic from error-handling logic. However, incorrect or imprecise use of these constructs may introduce faults that lead to unexpected behavior, crashes, or inconsistent program states.

[2] https://github.com/exception-handling/exception-handling-bugs

2.1. Exception handling anti-patterns

Anti-patterns are common mistakes in software development that tend to produce more negative consequences than benefits [19]. In the context of exception handling, exception handling anti-patterns refer to improper uses of exception handling mechanisms that may introduce bugs and reduce code maintainability [8,20,21]. Prior studies have documented several recurring exception handling anti-patterns across programming languages [16,21–23], such as overly broad exception catching, swallowing exceptions, and bare `except` blocks. In this study, we analyze a set of exception handling anti-patterns identified in prior research and commonly reported by static analysis tools and linters. The detailed definitions and detection strategies are described in Section 3.5.

2.2. Exception handling bugs and root causes

To precisely characterize when a defect should be considered an exception handling bug, we adopt the widely accepted definition proposed by Ebert et al. [13]. According to this definition, an exception handling bug is a defect whose cause is related to exception handling mechanisms, occurring when exceptions are incorrectly defined, thrown, propagated, handled, documented, or when they should have been thrown or handled but are not. This definition emphasizes that exception handling bugs are identified by their causal relation to exception handling mechanisms, rather than by the specific failures they produce. In practice, these bugs often manifest through observable consequences such as program crashes, resource leaks, or incorrect outputs. Studying their root causes is therefore essential, as they represent the underlying decisions that lead to such failures, for example, when an unhandled exception results in unexpected application termination.

3. Study design

Following the Goal Question Metric (GQM) template [24], the goal of this study is to investigate the root causes and fixes of exception handling bugs in Python projects. We also analyze the occurrence of exception handling anti-patterns and the impact of exception handling bugs fixes, particularly regarding the addition or removal of anti-patterns. To this end, we formulate the following research questions.

RQ_1. How often do exception handling anti-patterns, occur in Python projects?

Purpose: Assess the prevalence of exception handling anti-patterns in Python projects, highlighting which patterns pose the most significant risks to code quality and maintainability.

Metric: The frequency of each anti-pattern relative to the total identified.

RQ_2. What are the root causes of exception handling bugsin Python projects?

Purpose: Determine the most common underlying root causes of exception handling bugs. This understanding facilitates more effective bug detection, localization, and prevention.

Metric: Frequency of root causes (e.g., unhandled exceptions, incorrect exception types).

RQ_3. How are exception handling bugs fixed in Python projects?

Purpose: Identify recurring strategies used by developers when fixing exception handling bugs, enabling the understanding of effective fixing practices.

Metric: Frequency of fix patterns to remove exception handling bugs(e.g., adding exception handling blocks, changing exception types, adding raise conditions).

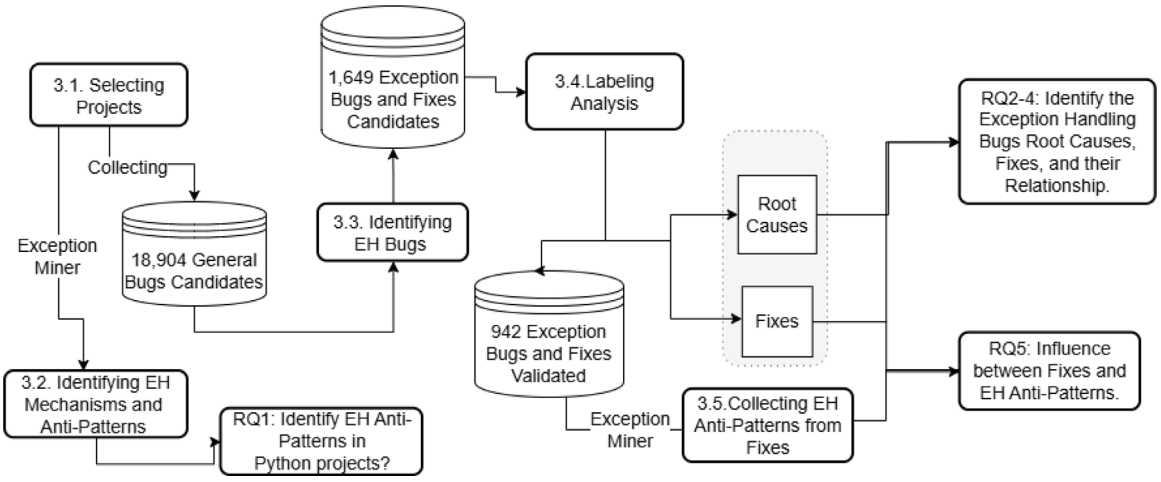


Fig. 1. Study design steps.

RQ_4. What is the relation between fixes and root causes of exception handling bugs?

Purpose: Reveal how specific root causes are addressed through corresponding fixes, helping developers match bug characteristics with effective repair strategies.

Metric: Percentage mapping between root causes and the fixes applied.

RQ_5. Does exception handling bug fixing influence the occurrence of exception handling anti-patterns?

Purpose: Evaluate whether fixing exception handling bugs reduces or introduces anti-patterns, highlighting risks associated with repair activities.

Metric: Frequency of anti-patterns added or removed during exception handling bugs fixes.

Fig. 1 provides an overview of the research methodology used to answer these questions. It illustrates the study workflow from project selection to the analysis steps corresponding to each research question.

3.1. Selecting projects

To perform our study, we used the SEART GitHub Search Engine [25][3] to identify Python open-source projects hosted on GitHub. Following criteria adopted in prior studies [10,21], we selected projects that: (i) are open-source and primarily developed in Python; (ii) are actively maintained, with at least one commit in the last three months and a history of at least three years; (iii) have significant community engagement (more than 10 contributors and 1k stars); and (iv) contain a sufficient number of reported issues (more than 50) using GitHub's issue tracking system. These criteria ensure that the selected projects are mature, actively maintained, and contain sufficient bug-related information for analyzing exception handling bugs.

A complete list of projects, including metadata such as commits, contributors, stars, and issues, is available on our accompanying webpage [26]. In total, our dataset includes 550 Python projects, with an average of 4109 commits, 127 contributors, 16,049 stars, and 1543 issues. To mitigate potential domain bias, we classified each project by its primary application domain using repository descriptions, GitHub topics, and project documentation. The dataset covers a diverse set of domains within the Python ecosystem, including Data Science, Machine Learning & AI (168 projects, 30.52%), DevOps, Cloud & Infrastructure (89 projects, 16.23%), End-user Applications & Systems (89 projects, 16.23%), Developer Tools & Libraries (89 projects, 16.24%), Web Development & APIs (64 projects, 11.69%), Security, Networking & Reverse Engineering (43 projects, 7.79%), Scientific & Numerical Computing (6 projects, 1.30%).

3.2. Extracting exception handling mechanisms

To extract exception handling mechanisms, we developed Exception Miner [27], a tool that analyzes Python abstract syntax trees (ASTs) to identify exception handling constructs such as `try-except`, `raise`, `try-finally`, and `try-else`. Exception Miner identifies these constructs by traversing explicit AST nodes rather than by relying on textual matching, keywords, or regular expressions.

The tool uses the Python binding of *Tree-sitter* [28] to define AST queries over Python syntactic structures. Specifically, `try-except`, `try-finally`, and `try-else` mechanisms are extracted from `try_statement` nodes, while `raise` mechanisms are extracted from `raise_statement` nodes. Applying these queries, we identified 168,716 exception handling mechanisms across 1,295,519 functions, including 58,186 `try-except`, 98,771 `raise`, 7,725 `try-finally`, and 4,034 `try-else` constructs. These numbers indicate that the analyzed projects contain a reasonable amount of exception handling mechanisms. To the best of our knowledge, our study is the largest study focused on exception handling bugs in Python. The list of AST queries to detect the aforementioned exception handling mechanisms, along with our tool, is available on the accompanying web page [26].

To evaluate whether the extracted exception handling mechanisms were affected by false positives, we manually validated a stratified sample of 384 mechanisms (95% confidence level, 5% margin of error). The sample includes instances of the four mechanism types analyzed in this study: `try-except`, `raise`, `try-finally`, and `try-else`. These mechanisms correspond to explicit Python AST structures: `try-except`, `try-finally`, and `try-else` are extracted from `try_statement` nodes, while `raise` mechanisms are extracted from `raise_statement` nodes. To do this, two authors independently inspected each sampled mechanism and its surrounding context to verify that the extracted snippet corresponded to the intended Python exception handling mechanism. Disagreements were resolved through discussion until a consensus was reached. The extraction step using our tool achieved 99.48% accuracy, 100.00% precision, 98.97% recall, and 99.48% F1-score, with detailed results by mechanism type available in the supplementary material [26].

[3] https://seart-ghs.si.usi.ch/

3.3. Collecting exception handling bugs and fixes

Unlike prior work that uses exception-related keywords to identify exception handling bugs in textual artifacts [14], our study does not use keywords as a proxy for exception handling semantics. Instead, following established mining software repositories [29,30], we use generic bug-related keywords only to retrieve candidate bug-fixing activities. Specifically, we mine the complete commit history of the analyzed Python projects and select commits whose messages contain ("*fix*" or "*solve*") and ("*bug*" or "*issue*" or "*problem*" or "*error*"). This process resulted in 18,904 candidate fixes from the 550 selected projects. These keywords are not used to classify commits as exception handling bugs; the final classification relies on code-level evidence, namely whether the fix modifies exception handling mechanisms, followed by manual validation.

After collecting the bugs and fixes, we identify the exception handling bugs (step 2 in Fig. 1) and associate them with their respective fixes by following these steps: First, we gather the issues or pull requests associated with previously identified fixes. Then, for each bug-fixing commit, we identify exception handling bug candidates by analyzing whether the fix modifies exception handling constructs in the source code. As most of the exception handling fixes are related to adding or removing exception handling mechanisms, we consider as a exception handling bug candidate, those that introduce changes within exception handling mechanisms, such as `try`, `except`, `raise`, `try-else`, or `try-finally`, following established approaches in prior empirical studies [29].

These candidates were automatically detected by traversing the AST and identifying modifications to constructs related to exception handling mechanisms. This preliminary step reduced the dataset to commits more likely to involve exception handling bugs, enabling focused manual validation. Using this process, we identified 1,649 candidates from the 18,904 bugs and pull requests collected in the previous stage. Since not all candidates correspond to genuine exception handling bugs, the first three authors manually inspected each case to confirm whether the observed bug was causally related to exception handling.

During validation, we examined the source code changes, issue or pull-request discussions, and, when available, related test cases to determine whether candidates corresponded to genuine exception handling bugs. In cases without explicit tests, we relied on issues labeled as "bug" and their corresponding fix commits as supporting evidence, following prior studies [14,15]. Candidates were excluded when the detected modification to an exception construct was incidental and unrelated to fixing an exception handling issue, such as general refactoring, non-bug-related updates (e.g., API adaptations or dependency updates), or false positives produced by the detection tool. For instance, in pull request #3271 from the PyTorch Lightning project,[4] the tool flagged a modification inside a `try-except` block, but manual inspection revealed that the actual fix addressed a different component unrelated to exception handling mechanisms. After applying this filtering process, we obtained 942 genuine exception handling bugs from the initial 1,649 candidates (942 confirmed bugs after removing 707-exception handling bugs candidates).

Our analysis also revealed that many fixes were accompanied by test cases designed to reproduce the exceptional behavior and validate the applied fix. In cases without explicit tests, we relied on issues labeled as "bug" and their corresponding fix commits as supporting evidence, following prior studies [14,15].

3.4. Labeling exception handling bugs, root causes, and fix patterns

In our study, we systematically categorize the root causes and fixes associated with exception handling bugs in Python projects (steps 3 and 4 in Fig. 1). Although prior studies propose taxonomies for classifying exception handling bugs [14,15], we derived our own taxonomy directly from empirical data rather than adopting predefined categories. This approach reduces potential bias in category selection and allows root cause and fix categories to emerge from the observed data. Despite this independent construction, our taxonomy shows similarities with existing classifications reported in previous studies [13,15], supporting its validity.

We adopted an open coding approach [24], commonly used in empirical software engineering studies [31], to label the root causes and fixes of exception handling bugs. We first conducted a pilot study on a random sample of 50 bugs from the total 1,649 exception handling bugs candidates. The first three authors independently analyzed these bugs without predefined categories and iteratively refined emerging patterns through discussion, resulting in an initial taxonomy of root causes and fix patterns.

Using this preliminary taxonomy, the remaining exception handling bugs were labeled by a team consisting of the three authors and five master's students with experience in Python programming and familiarity with exception handling mechanisms. Before starting the labeling task, the master's students participated in a training session. The training introduced the study context, Python exception handling mechanism, and the exception handling anti-patterns considered in the study. Participants also received a validation guide describing the step-by-step procedure: opening the bug report and fixing commit, checking whether the issue was actually related to exception handling, identifying root causes from the bug report and code review discussion, classifying the fix pattern from the code changes, and checking whether tests were added. The guide also instructed participants to disregard cases not related to exception handling and to refine or propose new categories when the initial taxonomy was insufficient.

The exception handling bugs were then divided into eight datasets and independently labeled by participants. Disagreements were resolved through group discussions, and the training and validation guide was used to align the students' classifications with the pilot study conducted by the authors. This iterative process expanded the taxonomy with nine additional root cause categories and twelve new fix pattern categories, resulting in the final taxonomy used in our analysis.

Labeling Root Causes: The root cause refers to a source of a bug such that if it is removed, the bug is decreased or removed [17], which includes technical factors in the source code that directly contributed to the exception handling bug. To identify them, we analyzed how exception handling mechanisms were misused or implemented, examining the buggy source files associated with each bug. Also, bug reports and comments provided helpful context, with a focus on their descriptions and discussions to identify potential observable root causes.

Labeling Fix Patterns: Fix patterns capture the code modifications developers applied to resolve the exception handling bugs. To this end, we analyzed changes in commits or pull requests, focusing on edits to exception handling constructs described in Section 3.2. By examining the exact modifications (e.g., adding an exception type, restructuring a `try-finally` block), we derived recurring strategies that developers used to repair bugs. Other sources, such as bug report discussions, were used only to clarify intent; however, the classification itself relied exclusively on the code changes.

To evaluate the reliability of the manual labeling process, three participants independently labeled a statistically significant sample of 312 bugs (95% confidence level, 5% margin of error). Inter-rater agreement was assessed using mean pairwise Cohen's kappa [32,33] and Krippendorff's alpha [34], following recommendations from prior work [35]. The analysis yielded a Cohen's kappa of 0.5285 and a Krippendorff's alpha of 0.5247, indicating moderate agreement according to established standards for empirical software engineering datasets [36].

During this validation process, we also recorded the labeling time, which averaged 121.92 s (approximately two minutes) per bug, reflecting the complexity of analyzing exception handling bugs.

[4] https://github.com/Lightning-AI/pytorch-lightning/pull/3271/

3.5. Extracting exception handling anti-patterns

Previous studies [8,20,21] investigate several bad practices related to exception handling mechanisms. These studies suggest that exception handling anti-patterns can lead to unexpected behavior, disrupt the program's normal flow, and introduce bugs. Thus, they should be avoided when developing or evolving software. In our study, we focus on a set of exception handling anti-patterns based on three criteria: i) the risks that they bring to the system; ii) whether they could be detectable by a static analysis tool; iii) whether they have been analyzed in previous studies [9,15,20–22] or technical documentation.[5] As a result, we analyze the following exception handling anti-patterns:

- **Too Broad Raising:** Raising exceptions that are too general, lacking specificity. For example, using a *Generic* exception type. This can hide bugs and make debugging harder.
- **Too Broad Except:** Handling exceptions that are too general, lacking specificity;
- **Swallowing Exceptions** Doing nothing or logging insufficient information inside an exception handler potentially leads to silent errors. This exception handling anti-pattern is common in Python using the `try-except-pass` flow.
- **Nested Try-Except Blocks:** Using three or more nested try-except blocks, thus indicating complex exception handling logic;
- **Bare Raise Block:** A `raise` statement with no exception provided, i.e., re-raising the last active exception. Bare raises should be used only in `except` blocks;
- **Bare Except Catch Block:** An `except` block with no exception provided. Bare except catches all exceptions, including exceptions such as *SystemExit* or *KeyboardInterrupt*;
- **Bare Raise inside Finally:** Raising exceptions directly within a `finally` block, which may lead to unexpected states and conditions;
- **Try and Return:** Using a `try` block solely to return from a function indicates a potential design issue.

We used our tool to collect the selected exception handling anti-patterns (steps 3.2 and 3.5 in Fig. 1). The tool was first applied to the latest versions of each project (Section 3.1) to identify existing anti-patterns in exception handling. We then manually inspected the fixes associated with exception handling bugs to detect the introduction or removal of exception handling anti-patterns during bug fixes.

Each exception handling anti-pattern is detected through specific AST queries implemented in our tool, based on definitions from prior studies [22]. For example, the *Nested Try-except* anti-pattern was originally identified manually in previous work and later translated into an AST query to enable automated detection across projects. The complete list of queries is available in the accompanying webpage [26].

To evaluate the detection accuracy of the Exception Miner tool [27], we manually validated a stratified sample of 766 instances: 381 positive and 385 negative exception handling anti-patterns (95% confidence level, 5% margin of error). The positive set contains candidates detected by Exception Miner, whereas the negative set contains exception handling snippets for which no anti-pattern was detected. Following prior work [37], two authors independently assessed each case using four confidence levels: *low*, *moderate*, *high*, and *absolute*. Disagreements were resolved through discussion until a consensus was reached. Exception Miner achieved 96.87% accuracy, 93.70% precision, 100.00% recall, and 96.75% F1-score. Detailed precision and recall results by anti-pattern category are available in the supplementary material [26].

To analyze whether exception handling bug fixes affect the occurrence of exception handling anti-patterns, we performed a before-and-after analysis for each fixing commit. For each confirmed exception handling bugs, we analyzed the buggy version immediately before the fix and the fixed version after the fixing commit. We then applied Exception Miner to the exception handling constructs affected by the fix in both versions. This procedure allows us to distinguish anti-patterns that were already present before the fix from those added, removed, or preserved during the repair. Therefore, our analysis does not assume that every anti-pattern observed in the fixed version was introduced by the fixing activity.

[5] https://github.com/pylint-dev/pylint

Table 1
Distribution of exception handling anti-patterns in Python Projects.

EH Anti-Pattern	Count	Perc (%)
Too Broad Except	18.455	41.47%
Swallowing Exceptions	12.674	28.48%
Too Broad Raising	7.263	16.32%
Bare Except	5.153	11.58%
Nested Try	812	1.82%
Bare Raise Block	134	0.30%
Bare Raise Finally	12	0.03%

4. Results

In this section, we describe the results and answer the research questions.

4.1. RQ_1. How often do exception handling anti-patterns occur in Python projects

Exception handling mechanisms are common in Python programs, but developers may misuse them, leading to exception handling anti-patterns. Thus, this research question investigates the prevalence of exception handling anti-patterns in Python projects that could potentially lead to maintainability problems, such as architectural degradation and technical debt [8].

Table 1 reports the distribution of exception handling anti-patterns identified in the analyzed projects. The first column presents the exception handling anti-patterns, while the second and third columns describe the occurrence and the corresponding percentage.

The results show that the occurrence of exception handling anti-patterns ranges from 0.32% to 41.47% across the analyzed exception handling mechanisms. The exception handling anti-pattern *Too Broad Except* has the highest occurrence (41.47%). This result aligns with the frequent use of generic exception types in the analyzed projects, such as *Exception* (13.54%) and *RuntimeError* (13.13%), which encourage overly broad catch blocks. Such practices reduce the specificity of error handling, obscure the underlying causes of failures, and complicate debugging and recovery. Prior studies similarly report the high prevalence and negative impact of this exception handling anti-patterns on program comprehension and maintainability [10,20,21].

The second most common exception handling anti-pattern is *Swallowing Exceptions*, affecting 28.48% of the analyzed functions. This exception handling anti-pattern occurs when generic exceptions are caught but neither rethrown nor properly handled. Developers often suppress errors using empty handlers or `pass` statements inside `except` blocks, allowing execution to continue without addressing the exceptional condition [38]. Such practices hinder fault diagnosis and have been widely reported as one of the most problematic Python anti-patterns [39].

Furthermore, *Too Broad Raising* and *Bare Except* present similar occurrence rates of 16.32% and 11.58%, respectively. Both anti-patterns are associated with the use of generic exceptions: the first involves raising overly general exceptions such as *Exception* or *BaseException*, while the latter corresponds to catch blocks that intercept all exceptions, including those not typically intended to be handled (e.g., *SystemExit* or *KeyboardInterrupt*). The remaining anti-patterns—*Nested Try*, *Bare Raise Block*, and *Bare Raise Finally*—occur less frequently, with rates ranging from 0.03% to 1.82%.

Table 2
Exception handling bugs distribution by root causes.

Root Cause	Total	Perc. (%)
Unhandled Exception	477	50.64%
Incorrect Handling Action	206	21.87%
Missing Exception Raising Condition	99	10.51%
Improper Exception Message	62	6.58%
Incorrect Re-raising of Exceptions	19	2.02%
API-Related Exception Handling Errors	18	1.91%
Wrong Exception Type	18	1.91%
Unexpected Raising	14	1.49%
Wrong Raise Type	14	1.49%
Import-Related Exception	9	0.96%
Improper Finally Block Usage	3	0.32%
Python Version Compatibility Issue	3	0.32%
Grand Total	**942**	**100.00%**

Summary: Exception handling anti-patterns are widespread in Python projects, with *Too Broad Except*, *Swallowing Exceptions*, and *Too Broad Raising* being the most prevalent.

The high prevalence of exception handling anti-patterns, particularly overly broad handlers and exception swallowing, indicates that imprecise and generic error-handling strategies remain common in Python projects. These findings suggest that static analysis tools and coding guidelines should prioritize detecting exception handling anti-patterns, especially those associated with overly generic exception handling and improper exception raising.

4.2. RQ_2. What are the root causes of exception handling bugs in python projects?

This research question investigates the root causes of exception handling bugs identified in our study and their manifestation during program execution. Table 2 summarizes the root causes observed, reporting their descriptions along with the corresponding frequency and percentage of occurrences.

Unhandled Exception (50.64%). This root cause occurs when exceptions are raised but not properly handled, allowing them to propagate beyond their intended scope and often leading to abrupt program termination or unexpected control-flow interruptions. Such effects often lead to application crashes, inconsistent states, or failures to handle expected exceptional conditions. Determining which exceptions should be explicitly handled remains challenging for developers, as indiscriminately catching all exceptions is considered bad practice [20,22,40]. Issue #6227[6] from the Ansible project illustrates this issue, where an unhandled exception related to character encoding caused execution to fail unexpectedly.

Incorrect Handling Action (21.87%). This root cause arises from flawed logic inside exception handling blocks, where the handling code executes incorrect or incomplete recovery actions. In practice, such bugs often lead to inconsistent behavior, partial functionality, or misleading outcomes during execution. Common cases include improper return values, incorrect conditional checks, or faulty recovery logic that allows execution to continue in an invalid state. Issue #2762[7] from the OpenBB project illustrates this root cause: incorrect handling logic within a `try-except` block led to unintended execution behavior.

Missing Exception Raising Condition (10.51%). This root cause includes situations in which exceptional conditions occur, but no exception is raised to signal the error. As a consequence, failures may remain undetected or manifest later as inconsistent behavior, incorrect results, or downstream crashes. These cases typically stem from missing validation checks or incomplete error-detection logic. An example is issue #10185[8] from the NumPy project, where the absence of a necessary exception raising condition allowed invalid values to propagate through the system (*AssertionError* when comparing NaT's (not a time type in Numpy) with different types. Addressing challenges associated with missing raising conditions is essential to improving error-detection mechanisms.

Improper Exception Message (6.58%). This root cause involves exceptions that convey inaccurate, incomplete, or misleading information to developers. While execution may fail correctly, the lack of clear diagnostic information significantly increases debugging effort and maintenance cost. In practice, this often leads to confusion during failure analysis or to incorrect assumptions about the source of an error. Issue #1155[9] from the Scikit-learn project exemplifies this root cause, where a developer argues that need to change the error message, since it references a non-existent attribute, obscuring the true source of the failure: "You refer to an attribute named `self.tokenize` which does not exist. According to the if-else structure, I assume the correct attribute here is `self.analyzer`, right?".

Incorrect Re-raising of Exceptions (2.02%). This root cause occurs when an exception is caught and then rethrown without adding any additional information or context. The main challenge is ensuring that re-raised exceptions contribute meaningful insights for effective debugging. Developers must balance providing sufficient information to understand the issue while avoiding unnecessary verbosity (e.g., flooding the application loggers or cluttering the client with unnecessary messages). For example, the Robot project's issue #2217[10] demonstrates this scenario. Note that re-raising the exception (*DataError*) causes issues in the application because it is not handled properly, leading to immediate failure.

API-Related Exception Handling Errors (1.91%). This root cause arises from changes or misuse of external APIs that alter exception behavior or contracts. Such issues frequently lead to execution failures after dependency updates, build failures in continuous integration environments, or unexpected runtime errors. Issue #10776[11] in the PyTorch Lightning project demonstrates this root cause: changes to an external API led to exception handling bug that required code adaptation.

Wrong Exception Type (1.91%). This root cause involves using inadequate or incorrect exception handling mechanisms, such as wrong exception types, leading to ineffective recovery (e.g., the intended handler is bypassed), confusing error reporting, or unexpected termination when the raised exception is not captured by the selected handler. For instance, issue #13613[12] from the *Requests* project describes an exception handling bug in which a *HTTPError* was handled instead of the more general *RequestException*. As noted by a developer, "the solution is to have raise_for_status handle *RequestException* instead of just *HTTPError*", ensuring that failures are captured consistently and do not propagate unexpectedly.

Unexpected Raising (1.49%). This root cause refers to scenarios where exceptions are raised unnecessarily or in unintended code locations. These cases typically manifest as disruptions to normal execution, premature termination of a feature, or partial functionality, even when the program could proceed safely. Issue #4554[13] from the *Pandas* project illustrates this root cause, where a developer indicated that a `raise` condition triggering a *TypeError* should be removed, because it unnecessarily prevented valid execution paths.

Wrong Raise Type (1.49%). This root cause involves instances where exceptions are raised using an incorrect exception type, leading

[6] https://github.com/ansible/ansible/issues/6227
[7] https://github.com/OpenBB-finance/OpenBBTerminal/issues/2762
[8] https://github.com/numpy/numpy/issues/10185
[9] https://github.com/scikit-learn/scikit-learn/issues/1155
[10] https://github.com/robotframework/robotframework/issues/2217
[11] https://github.com/Lightning-AI/pytorch-lightning/issues/6745
[12] https://github.com/psf/requests/issues/496
[13] https://github.com/pandas-dev/pandas/issues/4554

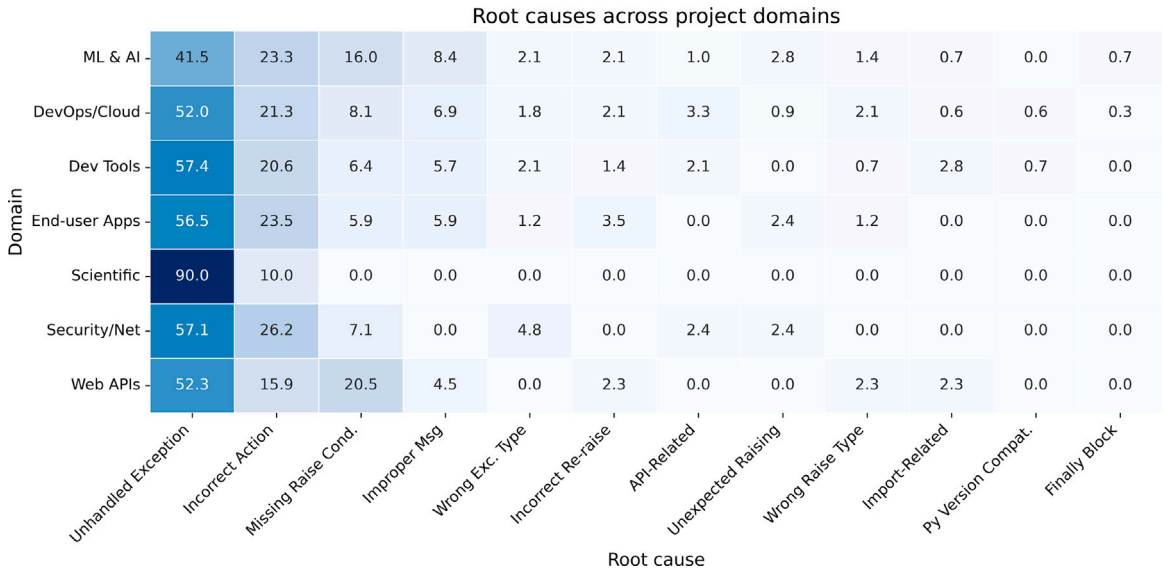


Fig. 2. Distribution of exception handling bug root causes across project domains.

to potential confusion during error handling. In practice, this can mislead developers during debugging, trigger inappropriate handling logic, or cause failures to be mishandled when callers catch a different exception type than the one raised. Consequently, error recovery becomes less reliable, and error diagnosis more difficult. Issue #13103[14] from the NumPy project exemplifies this scenario: the fix required changing a *ValueError* to a *TypeError* to better reflect the actual issue.

Import-Related Exception (0.96%). This root cause refers to errors encountered during module or dependency loading, where import failures are not handled appropriately or where exception handling around imports leads to inconsistent initialization behavior. In practice, these cases often manifest as crashes at startup, incomplete feature availability (modules silently missing), or failures in automated environments (e.g., CI) due to differences in dependencies. The Spyder project's bug report #2456[15] illustrates this scenario, where adding appropriate handling around imports prevents execution from failing during initialization and enables more robust recovery when dependencies are unavailable. This approach is common in Python due to the flexibility in handling module imports since developers can use the `try-except` clause to gracefully handle situations where imports may fail, making recovery possible for exceptional program execution.

Improper Finally Block Usage (0.32%). This root cause encompasses errors in cleanup logic associated with `finally` blocks, such as missing cleanup actions, incorrect cleanup execution, or inappropriate use of `finally`. In practice, these issues frequently manifest as resource leaks, inconsistent runtime state (e.g., unclosed database sessions or file handles), or performance degradation when resources accumulate across executions. Pull-request #8344[16] from the Freqtrade project exemplifies this root cause, where improper closing of a database session required correction to ensure resources are released correctly.

Python Version Compatibility Issue (0.32%). This root cause arises when exception handling behavior or code constructs are incompatible with a particular Python version or dependency configuration. These cases typically manifest as failures during execution or initialization, CI pipeline breakages, or inconsistent behavior across environments due to version-specific exception types. Issue #655[17] from the Pytest framework illustrates this scenario, where developers needed to "work around different ways that cause Python 2/3 causing failures in 3rd party code".

Summary: We identify 12 root causes associated with exception handling bugs. The most frequent root causes are *Unhandled Exception* and *Incorrect Handling Action*, which account for the majority of cases and are responsible for a wide range of observable execution-level failures, including abrupt termination, incorrect recovery behavior, and inconsistent program states.

The findings highlight the need for stronger testing practices that explicitly exercise exceptional execution paths, particularly those related to imports, API usage, and interpreter compatibility. The emergence of Python-specific root causes, such as import-related exceptions and version incompatibilities, also underscores the importance of multi-version testing and systematic dependency validation. For researchers, these results motivate the development of targeted mutation operators and testing strategies designed to expose faults in these scenarios.

4.2.1. Root causes across project domains

To investigate whether the distribution of root causes varies across application domains, we analyzed the frequency of each root cause within the projects of each domain. Fig. 2 presents a heatmap of this distribution, where each cell shows the percentage of root cause category in a given project domain.

The results show that *Unhandled Exception* remains the most frequent root cause in all analyzed domains, although its proportion varies across them. For example, it accounts for 41.5% of bugs in ML & AI, 52.3% in Web APIs, 52.0% in DevOps/Cloud, 57.4% in Developer Tools, and 56.5% in End-user Applications. Interestingly, ML & AI exhibits a lower proportion of *Unhandled Exception* and a higher proportion of *Missing Exception Raising Condition* (16.0%) compared with most other domains. Web APIs show a similar pattern, with *Missing Exception Raising Condition* accounting for 20.5% of bugs. These results suggest that ML/data pipelines and Web APIs often fail not only because exceptions are left unhandled, but also because invalid inputs, missing validations, or domain-specific preconditions are not explicitly signaled early through appropriate raise conditions.

4.2.2. Impact of exception handling bugs

To complement the root-cause taxonomy, we analyzed the observable impact of each confirmed exception handling bug when such

[14] https://github.com/numpy/numpy/issues/13103
[15] https://github.com/spyder-ide/spyder/issues/2456
[16] https://github.com/freqtrade/freqtrade/pull/8344
[17] https://github.com/pytest-dev/pytest/issues/655

Table 3
Observable impact of exception handling bugs. Percentages are computed over the 942 confirmed exception handling bugs.

Impact category	Description	Count (%)	Most frequent root causes
Incorrect Behavior	Program continues execution but produces wrong outcomes, such as incorrect recovery, unexpected control flow, or inconsistent data.	600 (63.7%)	Unhandled Exception (336); Incorrect Handling Action (128); Missing Exception Raising Condition (70)
Crash or Abrupt Termination	Program execution is unexpectedly interrupted due to an unhandled or improperly propagated exception.	116 (12.3%)	Unhandled Exception (72); Incorrect Handling Action (19); Missing Exception Raising Condition (11)
Misleading Diagnostics	Error messages are inaccurate, incomplete, or obscure the actual failure, hindering debugging and root-cause identification.	86 (9.1%)	Improper Exception Message (58); Unhandled Exception (13); Missing Exception Raising Condition (6)
Silent Failure	Exceptions are suppressed or handled too broadly, hiding the underlying problem while the program appears to continue normally.	82 (8.7%)	Incorrect Handling Action (37); Unhandled Exception (29); Missing Exception Raising Condition (7)
CI/Build Disruption	Exception handling defects cause build or test-suite failures, blocking continuous integration workflows.	34 (3.6%)	Unhandled Exception (16); Incorrect Handling Action (7); Missing Exception Raising Condition (5)
Performance Degradation	Improper exception handling introduces unnecessary overhead, such as catching and re-raising exceptions in hot paths.	16 (1.7%)	Incorrect Handling Action (8); Unhandled Exception (7); Wrong Exception Type (1)
Compatibility Failure	Exception handling code breaks across API versions, library updates, or Python versions.	8 (0.8%)	Unhandled Exception (3); API-Related Exception Handling Errors (2); Incorrect Handling Action (2)

information was available in issue reports, pull requests, commit messages, code review discussions, or associated tests. Table 3 summarizes the seven impact categories identified in our dataset, together with their frequency and most frequent associated root causes. It shows that *Incorrect Behavior* is by far the most common impact, accounting for 600 cases (63.7%). This indicates that exception handling bugs not only crash programs, but often allow execution to continue while producing incorrect outputs, inconsistent states, or unintended control flow. The second most frequent impact is *Crash or Abrupt Termination*, with 116 cases (12.3%), followed by *Misleading Diagnostics* (86 cases, 9.1%) and *Silent Failure* (82 cases, 8.7%). Less frequent impacts include *CI/Build Disruption* (34 cases, 3.6%), *Performance Degradation* (16 cases, 1.7%), and *Compatibility Failure* (8 cases, 0.8%). These results show that exception handling bugs affect execution correctness and reliability, but also hinder debugging, disrupt CI workflows, degrade performance, and expose compatibility problems. Representative examples for each category remain available in the supplementary material [26].

4.3. RQ_3. How are exception handling bugs fixed in python projects?

We next analyze how developers fix exception handling bugs. From the fixes associated with the bugs identified in our dataset, we derived 25 fix patterns grouped into ten categories using the open coding procedure described in Section 3. Table 4 summarizes these fix patterns, reporting the corresponding fixes and their frequency and percentage in the analyzed dataset.

Exception handling Block Operations (37.16%). This fix pattern group captures changes related to the introduction, removal, or relocation of exception handling blocks. It includes fixes such as *Add an exception handling block* (30.89%), *Remove an exception handling block* (4.14%), *Move the exception handling block* (1.49%), and *Add an exception handling block and raise an exception* (0.64%). These fixes reflect developers' efforts to explicitly handle exceptional conditions rather than allowing exceptions to propagate. For instance, pull request #860[18] in the Pytube library illustrates a case in which a developer introduces an exception handling block to enable graceful recovery from an exceptional condition, thereby preventing unexpected interruptions in program execution.

Exception Handling Type Operations (23.89%). This group comprises fixes that modify the exception types being handled, aiming to align exception handling with the actual failure conditions. It includes:*Change to an appropriate exception type* (14.65%), *Add an appropriate exception type* (8.92%), *Remove a specific exception type* (0.21%), *Add an appropriate exception type* (0.11%). Such fixes improve the precision of exception handling mechanisms and reduce the use of overly generic handlers. An illustrative example is pull request #58564[19] from the Ansible project, in which a developer adds handling for `OSError` to properly manage and ignore a specific error scenario.

Improve Raising Conditions (19.63%). This fix pattern group addresses refinements to the conditions under which exceptions are raised, ensuring that exceptions are triggered only when genuinely exceptional situations occur. The group includes Add a raise condition (16.45%), Change the raise type (1.80%), Remove a raise statement (1.17%), Move the raise expression (0.21%), and Change the exception type and raise an exception (0.11%). These fixes often prevent unnecessary disruption of control flow. For example, pull request #4304[20] from the Pandas project introduces a `raise ValueError` statement to explicitly and consistently signal invalid input.

Change the Exception Code Logic (10.08%). This group includes fixes in which developers restructure the surrounding logic of exception handling code, often using exceptions as part of the program's control flow. Such fixes are particularly common in Python, where developers frequently adopt the EAFP (Easier to Ask for Forgiveness than Permission) coding style [41]. Pull-request #273[21] from the AWS CLI project exemplifies this pattern, where the logic is adjusted by introducing additional checks inside a `try-except` block to ensure correct parsing behavior.

Change the Message Error (5.41%). This fix pattern focuses on improving the clarity and informativeness of error messages associated with exceptions. Clear error messages support debugging, log inspection, and the effective use of third-party libraries. Pull request #7440[22] from the PyTorch Geometric library illustrates this pattern, where a developer refines the error message produced when invalid values are passed to the `LinkNeighborLoader` class.

Re-raise an Exception (1.17%). This group includes fixes in which developers explicitly re-raise a previously caught exception to preserve failure behavior while potentially adding contextual information. Best practices recommend logging or enriching the exception before re-raising it [42]. A representative example is the mitmproxy commit fix,[23]

[18] https://github.com/pytube/pytube/pull/860

[19] https://github.com/ansible/ansible/pull/58564

[20] https://github.com/pandas-dev/pandas/pull/4304/files

[21] https://github.com/aws/aws-cli/pull/273

[22] https://github.com/pyg-team/pytorch_geometric/pull/7440

Table 4
Exception handling bugs distribution by fixes.

Fix Pattern Group	Fix Pattern	Count	Percentage	Total Group Perc.
Exception handling block operations	Add an exception handling block	291	30.89	37.16
	Remove the exception handling block	39	4.14	
	Move the exception handling block	14	1.49	
	Add an exception handling block and raise an exception	6	0.64	
Exception handling type operations	Change to appropriate exception type	138	14.65	23.89
	Add appropriate exception type	84	8.92	
	Remove a specific exception type	2	0.21	
	Add appropriate exception type and add a raise condition	1	0.11	
Improve raising conditions	Add a raise condition	155	16.45	19.63
	Change the raise type	17	1.80	
	Remove a raise statement	10	1.06	
	Move the raise expression	2	0.21	
	Change the exception type and raise an exception	1	0.11	
Change the exception code logic	Change the exception code logic	95	10.08	10.08
Improve the message error	Improve the message error	51	5.41	5.41
Re-raise an exception	Re-raise an exception	11	1.17	1.17
Import with an exception handling code	Import with an exception handling code	11	1.17	1.17
EH Anti-Patterns	Add a Swallowing Exception	3	0.32	0.75
	Change exception to generic type	2	0.21	
	Remove the Swallowing Exception	1	0.11	
	Skip test using Swallowing Exception	1	0.11	
Improve the Finally Mechanism	Add a finally mechanism	3	0.32	0.43
	Remove finally mechanism	1	0.11	
Fix API version	Fix API call and/or version	2	0.21	0.21
Python Version Compatibility Issue	Workaround to fix Python 2/3 incompatibility errors	1	0.11	0.11
Grand Total		**942**	**100.00%**	**100.00%**

which demonstrates a developer who re-raises protocol-related exceptions to preserve the full stack trace, thereby improving debugging and error inspection.

Import with an Exception Handling Code (1.06%). This fix pattern involves surrounding module imports with exception handling logic to handle failures during initialization or dependency loading. This approach allows programs to recover gracefully when optional or environment-specific dependencies are unavailable. A pull request from the Ansible project demonstrates this pattern, in which a `try-except` block is added around an import to ensure compatibility with a newer API version.

EH Anti-Patterns (0.75%). This group captures fixes that introduce, modify, or remove exception handling anti-patterns, including *Add a Swallowing Exception* (0.32%), *Change exception to generic type* (0.21%), *Remove the Swallowing Exception* (0.11%), and *Skip test using Swallowing Exception* (0.11%). Such patterns are generally discouraged, as they obscure failures and complicate debugging. Pull request #5022[24] from the Pandas project illustrates a corrective action addressing the broad exception exception handling anti-pattern .

This fix pattern group includes changes to `finally` blocks to ensure proper cleanup and resource management. It includes *Add a finally mechanism* (0.32%) and *Remove finally mechanism* (0.11%). Pull request #8344[25] from the Freqtrade project exemplifies this pattern, where a `finally` block is introduced to reliably release concurrent resources.

Fix API version (0.21%). This group includes fixes applied when API version mismatches are identified as the root cause of exception handling issues. Developers typically adjust API usage and surround calls with appropriate exception handling. Pull request #7448[26] from the Pandas project illustrates a fix that updates the dependency API to restore correct exception behavior.

Python Version Compatibility Issue (0.11%). Developers apply this fix pattern group by addressing compatibility issues across Python versions by introducing version-specific workarounds in exception handling logic. Issue #655[27] from the Pytest library demonstrates such a case, where developers implement adjustments to accommodate differences between Python 2 and Python 3 exception handling.

> **Summary:** We identify 25 fix patterns for exception handling bugs. The most prevalent patterns *Add an exception handling block*, *Change to an appropriate exception type*, and *Add a raise condition* account for nearly 62% of all fixes, indicating that developers mainly address exception handling bugs by introducing explicit handling mechanisms and refining exception signaling.

The predominance of fix patterns such as adding exception handling mechanisms, refining raise conditions, and correcting exception types indicates that developers typically resolve exception handling bugs by making error handling more explicit and precise. This observation suggests that automated repair and recommendation tools could prioritize these dominant strategies as repair templates.

4.4. RQ_4. What is the relation between fixes and root causes of exception handling bugs?

After analyzing the fix patterns applied to exception handling bugs, we examined whether repair strategies vary across root cause categories. Fig. 3 presents the normalized co-occurrence matrix relating 12 root causes to 11 fix pattern groups across 942 instances. A chi-square test of independence rejects the null hypothesis ($\chi^2(110, N=942) = 2007.77$, $p < 0.001$), indicating a moderate-to-strong association between root causes and fix patterns.

[23] https://github.com/mhils/mitmproxy/commit/b768b51327821d7d8db83dfe2362da3896059703
[24] https://github.com/pandas-dev/pandas/pull/5022
[25] https://github.com/freqtrade/freqtrade/pull/8344
[26] https://github.com/pandas-dev/pandas/pull/7448
[27] https://github.com/pytest-dev/pytest/issues/655

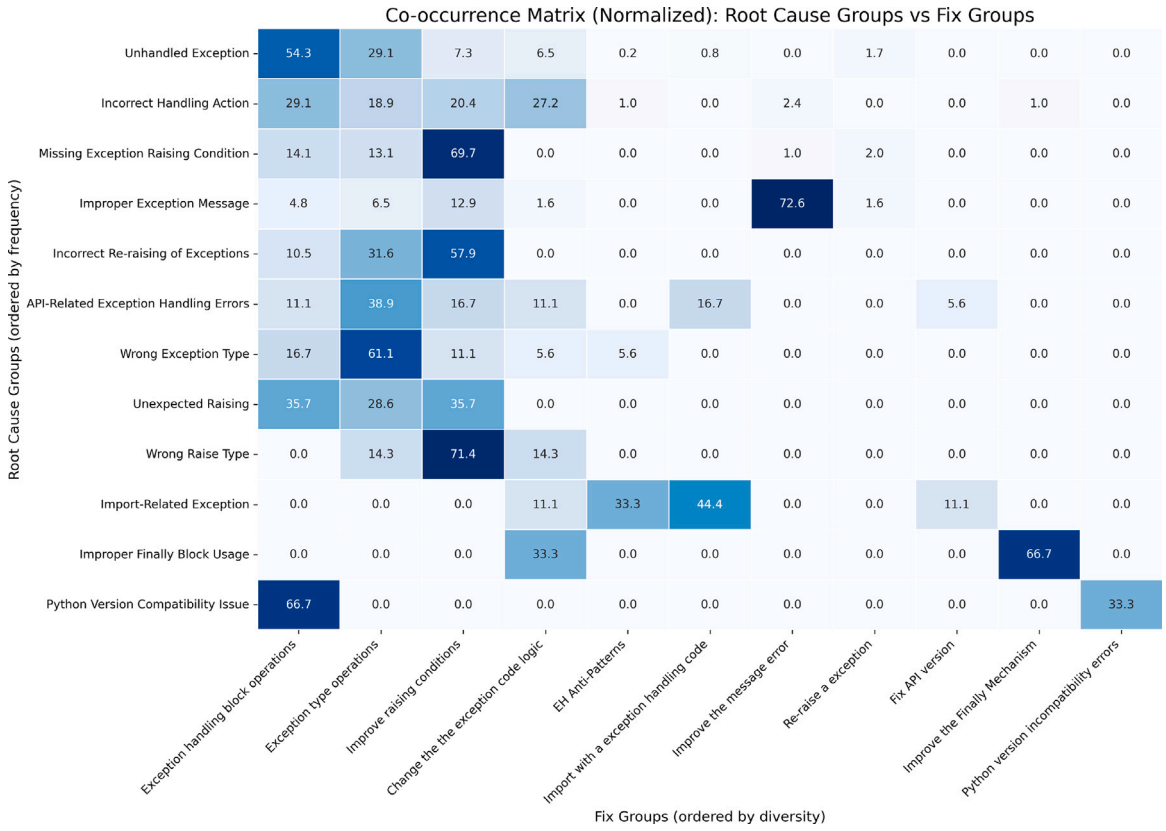


Fig. 3. Distribution of fix patterns group and root causes.

To quantify the strength of this relationship, we report the bias-corrected Cramer's V, which yields 0.45 (95% confidence interval [0.38, 0.55]), corresponding to a moderate-to-strong association. This effect size indicates that repair actions are not evenly distributed across root causes; instead, several root causes are associated with dominant fix strategies, suggesting that developers tend to apply targeted repairs aligned with the underlying cause of the exception handling bugs.

Unhandled Exception. Developers primarily fix this root cause using *Exception handling block operations*, which account for 54.3% of the fixes. Other strategies include *Exception type operations* (29.1%), *Improving raise conditions* (7.3%), and *Changing the exception code logic* (6.5%), while the remaining fixes account for 2.7%. Fig. 4(a) illustrates an example patch where a `try-except` block is added to handle a *UnicodeEncodeError*. This relationship is also supported by the standardized residual analysis (residual = 6.14), indicating that *Exception Handling Block Operations* occur substantially more often than expected for this root cause.

Incorrect Handling Action. Developers fix this root cause using several strategies, primarily *Exception handling block operations* (29.1%), followed by *Change the exception code logic* (27.2%) and *Improving raise conditions* (20.4%). Other fixes include *Exception type operations*, *EH anti-pattern corrections*, *Improve message error*, and *Improve the finally mechanism*. Fig. 4(b) shows a patch fixing the *Incorrect Handling Action* root cause scenario where the logic execution order is incorrect, thus leading to the error. Standardized residual analysis shows a strong association between *Incorrect Handling Action* and *Change the Exception Code Logic* (residual = 7.73), indicating that developers often resolve this root cause by restructuring exception-related logic rather than modifying the exception itself.

Missing Exception Raising Condition. Developers primarily address this root cause by *Improving raising conditions* (69.7%), followed by *Exception handling block operations* (14.1%) and *Exception type operations* (13.1%). The remaining fixes account for 3.0%. Fig. 4(c) illustrates a

(a) Unhandled Exception (Ansible#6277)

```
- data = data.decode('utf-8')
+ if isinstance(data, unicode):
+    try:
+       data = data.decode('utf-8')
+    except UnicodeEncodeError, e:
+       pass
```

(b) Incorrect Handling Action (OpenBB#237)

```
  except Exception:
-    lunchbreak_end = None
-    lunchbreak_start = None
+    after_lunch_start = False
+    before_lunch_end = False
```

(c) Missing Raising Condition (Numpy#10185)

```
- return
+ if dtypes_match:
+    return
+ else:
+    raise AssertionError(msg)
```

(d) Improper Exception Message (Scikit-learn#1155)

```
- raise ValueError('%s is not a
-     valid tokenization scheme'
-     % self.tokenize)
+ raise ValueError('%s is not a
+     valid tokenization
+     scheme/analyzer' %
+     % self.analyzer)
```

Fig. 4. Different root causes and their respective fixes in exception handling bugs.

(e) Incorrect Re-raising of Exceptions (Robot#2217)

```
- raise DataError('Replacing
-     variables from keyword
-     return value '
-     'failed: %s' % err.message)
+ raise VariableError('Replacing
+     variables  from keyword
+     return value '
+     'failed: %s' % err.message)
```

(f) API-Related Exception Handling Errors (Pytorch#10776)

```
except ImportError:
    _MLFLOW_AVAILABLE = False
-   mlflow, MlflowClient = None, None
+   mlflow, MlflowClient,
+     context = None, None, None
```

(g) Wrong Exception Type (Request#13613)

```
- except HTTPError:
+ except RequestException:
    return False
```

(h) Unexpected Raising (Pandas#4554)

```
- if order not in ['C', 'F']:
-     raise TypeError(
-       "must specify a tuple /
-       singular length to reshape")
```

(i) Wrong Raise Type (Numpy#13103)

```
- raise ValueError("cannot convert
-     'to_end' to array with
-     dtype" "'%r' as required
-     for input ary" % dtype_req)
+ raise TypeError("dtype of `to_end`
+     must be compatible
+     with input `ary` under
+     the `same_kind` rule.")
```

(j) Import-Related Exception (Spyder#2456)

```
- except ImportError:
+ except:
    pd = None
```

(k) Improper Finally Block Usage (Freqtrade#8344)

```
- yield _rpc
- Trade.rollback()
+ try:
+     yield _rpc
+ finally:
+     Trade.session.remove()
+     _request_id_ctx_var
+     .reset(ctx_token)
```

l) Python Version Compatibility Issue (Pytest#655)

```
- PYTHONPATH=os.path.dirname(
-   ANSIBLE_LIB_ROOT)
+ PYTHONPATH=asd()
+ try:
+    return (
+     asd.python_path)
+ except AttributeError:
+    pass
```

Fig. 4. (*continued*).

patch where a missing `raise` statement is added. This relationship is reinforced by the standardized residual analysis (residual = 11.24), indicating a strong association between this root cause and the *Improve Raising Conditions* fix pattern.

Improper Exception Message. Developers primarily fix this root cause by *Improving the exception message* (72.6%). Other strategies include *Improving raising conditions* (12.9%), *Exception type operations* (6.5%), and *Exception handling block operations* (4.8%), while *Re-raising an exception* accounts for 1.6%. Fig. 4(d) illustrates a patch improving the exception message. The standardized residual analysis (residual = 22.73) shows the strongest association observed in our study, indicating that developers overwhelmingly address this root cause by refining exception messages rather than applying alternative fixes.

Incorrect Re-raising of Exceptions. Developers primarily address this root cause by *Improving raising conditions* (57.9%), followed by *Exception type operations* (31.6%) and *Exception handling block operations* (10.5%), reinforcing the importance of proper exception handling and raising mechanisms to ensure effective re-raising of exceptions. Fig. 4(e) illustrates a patch in which *DataError* is changed to *VariableError* to correct the re-raised exception.

API-Related Exception Handling Errors. Developers primarily fix this root cause using *Exception type operations* and *Exception handling block operations*, which together account for 50% of the fixes. Other strategies include *Improving raising conditions* and *Import with an exception handling code* (33.4%), followed by *Change the exception code logic* (11.1%) and *Fix API version* (5.6%). Fig. 4(f) illustrates a patch introducing new tags required by a newer MLFlow API version.

Wrong Exception Type. Developers primarily fix this root cause using *Exception type operations* (61.1%), followed by *Exception handling block operations* (16.7%), *Improve raising conditions* (11.1%), *EH anti-pattern corrections* (5.6%), and *Change the exception handling code* (5.6%). Fig. 4(g) illustrates a patch correcting the exception type.

Unexpected Raising. Developers fix this root cause using *Improve raising conditions* (35.7%), *Exception handling block operations* (35.7%), and *Exception type operations* (28.6%). Fig. 4(h) illustrates a fix from Pandas where an unnecessary `raise` statement is removed that involves applying fixes to improve raising mechanisms.

Wrong Raise Type. Developers fix this root cause mainly through *Improve raising conditions* (71.4%), followed by *Exception type operations* (14.3%) and *Change the exception code logic* (14.3%). These fixes emphasize the importance of using the correct raise types in the code, thereby addressing cases where incorrect raise types were used. Fig. 4(i) shows a patch in which *ValueError* is changed to *TypeError* to correct the raised exception type.

Import-Related Exception. Developers mainly fix this root cause using *Import with exception handling code* (44.4%), followed by *EH anti-pattern corrections* (33.3%), and both *Change the exception code logic* and *Exception type operations* (11.1% each). Fig. 4(j) illustrates a patch adding an `except` block to prevent application crashes during an import, demonstrating how developers face challenges specific to importing modules in Python. This association is reinforced by a large standardized residual (12.01), indicating a consistent fix strategy for import-related failures.

Improper Finally Block Usage. Developers fix this root cause mainly by *Improving the finally mechanism* (66.7%) and, less frequently, by *Changing the exception code logic* (33.3%). Fig. 4(k) illustrates a patch, which consists of adding the `finally` block. The `finally` block is essential to ensure robust cleanup procedures in critical sections of the codebase (database connections, file handling, and other resource-intensive tasks.

Python Version Compatibility Issue. Developers fix exception handling bugs with this root cause using *Exception handling block operations* (66.7%) and *Python version incompatibility fixes* (33.3%). The first denotes the primary strategy, whereas the latter directly addresses issues arising from differences across Python versions. Fig. 4(l) illustrates an example patch resolving this compatibility issue.

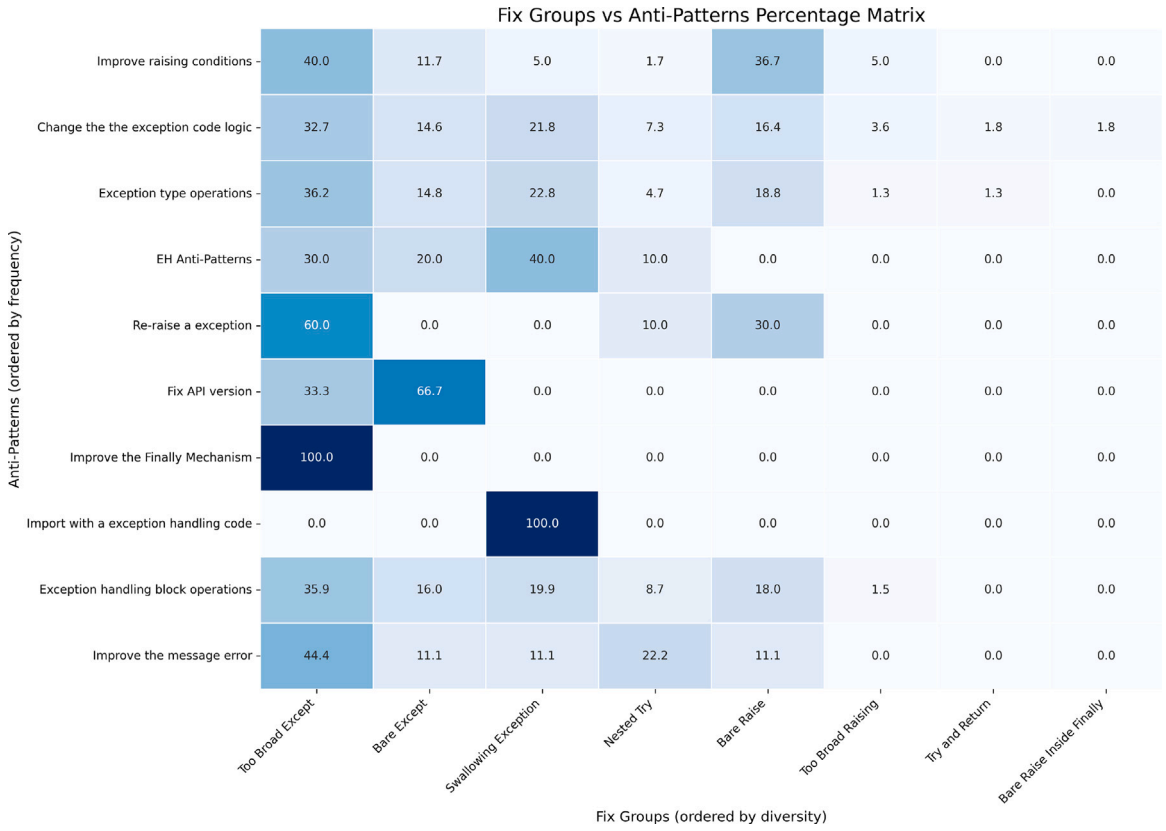


Fig. 5. Distribution across exception handling fixes and anti-patterns.

Summary: We observe a statistically significant and moderate-to-strong association between exception handling root causes and fix patterns. This indicates that developers tend to apply targeted fixes aligned with the underlying causes of exception handling bugs. The strongest associations occur for the root causes *Missing Exception Raising Condition*, *Improper Exception Message*, *Unhandled Exception*, and *Import-Related Exception*.

The results show that several root causes are associated with highly concentrated repair strategies, indicating that identifying the underlying root cause can effectively guide fix recommendations. This observation opens opportunities for automated detection and repair tools to provide more targeted interventions. For instance, tools could recommend introducing handling blocks for unhandled exceptions, refining raise conditions when exception signaling is incomplete, or improving diagnostic messages when error communication is insufficient. From a research perspective, these structured mappings between root causes and fixes provide a valuable foundation for developing predictive and learning-based repair models.

4.5. RQ_5. *Does exception handling bug fixing influence the occurrence of exception handling anti-patterns?*

Beyond analyzing the prevalence of exception handling anti-patterns in RQ_1, we examined whether specific fix patterns are associated with the introduction or modification of anti-patterns. Fig. 5 presents the normalized co-occurrence matrix relating 10 fix groups to 8 exception handling anti-patterns categories, enabling us to analyze how different fixes contribute to the introduction of exception handling anti-patterns.

Improve Raising Conditions. This fix pattern group addresses changes related to raising conditions, including adding, removing, or modifying `raise` statements. It introduces several anti-patterns, most notably *Too Broad Except* (40.0%) and *Bare Raise* (36.7%), highlighting challenges in selecting appropriate raising conditions. Other exception handling anti-patterns are introduced less frequently, including *Bare Except* (11.67%).

Change the Exception Code Logic. This fix pattern group is associated with several introduced exception handling anti-patterns, including *Too Broad Except* (32.7%), *Swallowing Exception* (21.8%), *Bare Raise* (16.4%), and *Bare Except* (14.6%). These results highlight the complexity of modifying exception-related logic, indicating that while this fix may address specific logic issues, it introduces diverse exception handling anti-patterns.

Exception Type Operations. This fix pattern group introduces several exception handling anti-patterns, most notably *Too Broad Except* (36.2%). Other predominant anti-patterns include *Swallowing Exception* (22.8%), *Bare Raise* (18.79%), and *Bare Except* (14.8%). Although these fixes aim to improve the robustness of exception handling by modifying exception types, they may inadvertently introduce anti-patterns during the repair process.

EH Anti-Patterns. This fix pattern group targets issues related to the presence of exception handling anti-patterns. Our analysis shows that although these fixes resolve specific issues, they may inadvertently introduce new exception handling anti-patterns. In particular, *Swallowing Exception* accounts for 40.0% of the introduced anti-patterns, followed by *Too Broad Except* (30.0%), *Bare Except* (20.0%), and *Nested Try* (10.0%). This result aligns with prior studies [20] highlighting the challenges of refactoring exception handling anti-patterns, where resolving one issue may unintentionally introduce another.

Re-raise an Exception. This fix pattern group, centered around re-raising exceptions, introduces specific anti-patterns during its application. With *Too Broad Except* representing 60.0%, *Bare Raise* representing 30.0%, and *Nested Try* representing 10.0% of the exception handling anti-patterns.

Fix API Version. This fix pattern group addresses issues related to API version compatibility or errors. These fixes contains exception

Table 5
Before-and-after analysis of exception handling anti-patterns in bug-fixing commits.

EH anti-pattern	Before	After	Delta	% change
Too Broad Except	2574	2685	+111	4.31%
Swallowing Exceptions	1798	1851	+53	2.95%
Too Broad Raising	739	753	+14	1.89%
Bare Except Catch Block	658	680	+22	3.34%
Nested Try-Except Blocks	48	49	+1	2.08%
Bare Raise Block	26	21	−5	−19.23%
Bare Raise inside Finally	1	1	0	0.00%
Total	**5,844**	**6,040**	**+196**	**3.35%**

handling anti-patterns such as *Bare Except* (66.7%) and *Too Broad Except* (33.3%). These anti-patterns underscore the trade-offs of ensuring API version compatibility, suggesting that developers must carefully manage potential anti-patterns introduced during the process.

Improve the Finally Mechanism. The fix pattern group addresses issues within the *finally* mechanism, i.e., Add or remove a *finally* mechanism. This fix may introduce the *Try and Return* exception handling anti-pattern in 100% of the fixes analyzed.

Import With an Exception Handling Code. The fix pattern group is designed to handle import-related issues. However, its application introduces the *Swallowing Exception* exception handling anti-pattern, which represents 100%. This highlights a potential downside of this fix, underscoring the importance of developers being aware during import-related fixes.

Exception Handling Block Operations. This fix pattern group modifies existing exception handling blocks to remove bugs, including adding, removing, or relocating blocks. These fixes may introduce several exception handling anti-patterns (six in total), most notably *Too Broad Except* (35.9%), followed by *Swallowing Exception* (19.9%), *Bare Raise* (18.0%), and *Bare Except* (16.0%). Although these fixes improve exception handling mechanisms, developers should remain aware of the potential introduction of anti-patterns. For example, pytest PR #786 fixes a bug in which `PrettyPrinter.pformat()` fails when formatting sets containing unsortable elements. The fix adds a `try-except` block around the formatting logic and falls back to using `repr()` when formatting fails. Although this change prevents failures and improves the robustness of the formatting behavior, the added handler catches the generic `Exception`, which violates the *Too Broad Except* anti-pattern.

Improve the Message Error. This fix pattern group focuses on refining error messages. While addressing message-related issues, it may introduce exception handling anti-patterns such as *Too Broad Except* (44.4%), *Nested Try* (22.2%), *Swallowing Exception* (11.1%), *Bare Raise* (11.1%), and *Bare Except* (11.1%). These results suggest that message-related fixes often occur alongside existing exception handling practices rather than explicitly removing or adding exception handling anti-patterns.

To investigate whether the occurrence of exception handling anti-patterns depends on the type of fix applied, we performed a chi-square test of independence between fix groups and anti-pattern categories. The test did not reject the null hypothesis ($\chi^2(63, N=504) = 67.07$, $p = 0.34$), indicating that the distribution of exception handling anti-pattern categories does not significantly differ across fix groups. Consistent with this result, the bias-corrected Cramer's V was 0.034 (95% bootstrap CI [0.025, 0.150]), indicating a negligible association. We also performed a before-and-after analysis to distinguish exception handling anti-patterns that were already present before the fix from those added or removed during the repair. Table 5 reports the detailed before-and-after counts for each exception handling anti-pattern category. Positive delta values indicate anti-pattern occurrences added after the fix, whereas negative values indicate removals.

The analysis shows that many exception handling anti-patterns observed after fixing were already present in the buggy version and preserved after the repair. However, the total number of anti-pattern occurrences increased from 5844 before the fixes to 6040 after the fixes, a net increase of 196 occurrences, corresponding to a 3.35% increase. The largest absolute increases occurred for *Too Broad Except* (+111), *Swallowing Exceptions* (+53), and *Bare Except Catch Block* (+22), whereas *Bare Raise Block* decreased by five occurrences. These findings suggest that although exception handling fixes do not systematically introduce specific categories of exception handling anti-patterns, they may still preserve or slightly increase during bug fixing.

> **Summary:** Many exception handling anti-patterns were already present before the fix, but exception handling fixes still resulted in a net increase of 196 anti-pattern occurrences. Their occurrence is statistically independent of the applied fix strategy, indicating that anti-patterns may arise across diverse fixing activities. The most frequent case is *Too Broad Except*, which appears in seven of the ten fix groups.

These results refine our interpretation by showing that many exception handling anti-patterns were preserved from the buggy version rather than introduced directly by the fixes. However, the observed net increase indicates that exception handling fixes can still introduce maintainability risks, particularly through broad handlers and swallowed exceptions. Thus, developers, reviewers, and tools should mitigate this risk by incorporating change-aware analyses to detect exception handling anti-patterns during bug fixes and by integrating these checks into continuous integration pipelines to provide early feedback to developers.

5. Implications

Our findings from RQ_1 indicate that exception handling anti-patterns such as *Swallowing Exceptions* and overly broad handlers (*Too Broad Except*) remain highly prevalent in Python projects. This observation aligns with prior studies on Java that report widespread use of generic catch blocks and silent exception suppression, both of which are associated with reduced reliability and maintainability [8,15]. In Python, excessive use of generic exception handling can obscure the root cause of failures and hinder debugging. Therefore, developers should prioritize more precise exception typing and informative error reporting, while tool support should emphasize the detection of exception handling anti-patterns that compromise fault diagnosis.

Our analysis of root causes (RQ_2) provides insights for identifying common exception handling bugs in Python projects. Table 6 compares our root cause taxonomy with related concepts reported in prior exception handling studies. Extending prior studies on exception handling bugs in Java, Android, cloud systems, and exception handling anti-patterns [8,13,15,29,43,44], we observe that several root causes have strong conceptual counterparts in prior work, such as *Unhandled Exception*, *Incorrect Handling Action*, *Wrong Exception Type*, and *Improper Finally Block Usage*. At the same time, we identify root causes that are especially relevant in the Python ecosystem, such as *Import-Related Exception* and *Python Version Compatibility Issue*. These categories are closely related to Python's runtime import model and optional dependency handling. To mitigate these issues, developers should strengthen

Table 6
Taxonomy-level comparison between the root causes identified in our Python study and related concepts reported in prior exception handling studies.

Root Cause	Related concepts in prior studies	Supporting studies	Similarity	Python-specific aspect
Unhandled Exception	Missing handler, uncaught exception, unhandled runtime exception, unhandled exception anti-pattern	[8,13,15,29,43,44]	Strong	Frequent due to runtime failures and lack of checked exceptions; absence of handling is only exposed during execution.
Incorrect Handling Action	Faulty handler behavior, incorrect recovery action, inappropriate fallback, handler logic faults	[13,15,29,43,44,46]	Strong	Often related to Python's EAFP style, where handlers are part of normal control-flow decisions.
Missing Exception Raising Condition	Missing error signaling, missing validation, failure to report exceptional state	[15,47,48]	Moderate	Particularly relevant in Python because invalid states may propagate dynamically until later runtime failures.
Improper Exception Message	Inconsistent or misleading diagnostic information; inaccurate exception/message relation	[47,49]	Moderate	Important in Python libraries, where exceptions and messages are often the primary API-level diagnostic mechanism.
Incorrect Re-raising of Exceptions	Incorrect propagation, destructive remapping, exception wrapping, loss of diagnostic context	[10,11,13,15,29]	Strong	Python traceback preservation and explicit re-raising semantics make this category especially relevant.
API-Related Exception Handling Errors	API-induced exception bugs, undocumented exceptions, framework-specific exceptions	[11,29,43,44,46]	Strong	Often caused by dependency updates and unstable runtime contracts in Python package ecosystems.
Wrong Exception Type	Wrong caught/thrown exception type; overly general or inappropriate exception selection	[10,11,13,15,49]	Strong	Not checked at compile time; handlers may fail to catch the intended exception only at runtime.
Unexpected Raising	Unnecessary throwing, overly defensive exception behavior, inappropriate exceptional control flow	[13,15,29,49]	Moderate	Can be linked to permissive runtime behavior and validation style in Python libraries.
Wrong Raise Type	Wrong thrown exception type, inappropriate exception construction	[13,15,29,49]	Strong	Python permits flexible runtime raising patterns, increasing dependence on developer discipline.
Import-Related Exception	Dependency/configuration failures, optional dependency failures, API/environment failures	[11,43,44,46,50]	Weak to moderate	Python-specific because imports are executable runtime statements, and optional imports are commonly guarded with `try-except` blocks.
Improper Finally Block Usage	Cleanup faults, resource-release errors, finally-related defects	[10,12,13,15,29]	Strong	Similar to other languages, Python provides an alternative construct for cleanup.
Python Version Compatibility Issue	Environment/version-related exception bugs, platform/runtime-specific exception behavior	[11,44,50]	Weak to moderate	Arising from Python 2/3 differences, interpreter behavior, and version-dependent library semantics.

testing practices, for example, by applying differential testing across multiple Python interpreters and environments. These findings also suggest opportunities to improve testing techniques by tailoring mutation operators to exception handling mechanisms. Building on prior work advocating exception mutation operators [45], we propose two operators targeting root-cause-specific scenarios: *Changing the Raise Type*, which modifies exception types to expose bugs related to the *Wrong Raise Type* root cause, and *Remove the Try Block Around Imports*, which removes try blocks surrounding import statements to reveal bugs associated with *Import-Related Exceptions*. These operators may help developers design more effective test suites for detecting exception handling bugs.

Our findings highlight opportunities to improve automated exception handling tools. Prior work has explored mining missing API error-handling specifications [51], detecting inconsistent exceptions and messages [47], recommending exception types [49], recommending handling locations [52], and generating exception handling code [53–55]. Recent work also explores machine learning and LLM-based support, such as Neural Exception Handling Recommender [56], Neurex [55], Seeker [57], and SHIELDA [58]. Our results complement these efforts by providing Python-specific evidence of root causes, fix patterns, and exception handling anti-patterns observed in real bug fixes. In particular, the prevalence of *Unhandled Exception*, *Missing Exception Raising Condition*, and *Improper Exception Message* suggests that future tools should detect missing handling and validation logic, recommend suitable exception types, and verify whether generated fixes introduce exception handling anti-patterns.

Regarding RQ_4, our results show that repair strategies are not applied arbitrarily. The chi-square test reveals a significant association between root causes and fix patterns with a moderate-to-strong effect size, indicating that developers tend to apply targeted repairs aligned with the underlying cause of the exception handling bugs. For instance, API-related root causes are often addressed by adjusting exception types or refining exception handling mechanisms, whereas missing raising conditions are typically resolved by introducing or refining `raise` statements. These associations suggest opportunities for rule-based or learning-based systems that recommend repair strategies based on the identified root cause.

In contrast, our analysis of fix patterns and anti-patterns (RQ_5) indicates that the occurrence of exception handling anti-patterns does

not significantly depend on the type of fix applied. The chi-square test revealed no statistically significant association and a negligible effect size, suggesting that anti-patterns are dispersed across fix categories rather than concentrated in specific repair strategies. Consequently, developers may introduce exception handling anti-patterns regardless of the repair strategy employed. This finding reinforces the need for independent detection mechanisms capable of identifying exception handling anti-patterns during maintenance activities and highlights opportunities for tools that prevent the introduction of anti-patterns during exception handling fixes.

6. Threats to validity

We discuss threats to validity in accordance with the guidelines of Wohlin et al. [59].

6.1. Construct validity

A potential threat concerns the accuracy of our tool for extracting exception handling anti-patterns. To mitigate this threat, we extensively tested and manually validated the tool (Section 3.5) and relied on a parser library [28] to detect patterns previously used in related studies [60,61]. Another threat is the omission of exception handling bugs that are not associated with commits retrieved via generic bug-related keywords [46]. In our approach, keyword filtering is used only to identify a broad set of bug-fixing activities. The classification of a bug as an exception handling bug is determined by whether the corresponding fix modifies exception handling mechanisms at the code level, as done in prior studies [29]. Therefore, even when exception handling is not explicitly mentioned in commit messages or issues, fixes that affect exception handling code are still captured.

Another threat concerns the use of keywords to retrieve candidate bug-fixing commits. Such keywords may miss fixes described with alternative, project-specific, vague, or implicit terminology [46]. We mitigate this threat by using keywords only as an initial retrieval mechanism, not as evidence that a bug is related to exception handling. Also, the final classification of an exception handling bug relies on code-level evidence and is followed by manual validation. Nevertheless, our dataset represents the universe of exception handling bugs accessible through our retrieval strategy and manual validation process, rather than an exhaustive collection of all possible exception handling bugs in the analyzed projects.

6.2. Internal validity

Our study does not include all exception handling anti-patterns reported in the literature; instead, we focus on the most prevalent ones that can be automatically detected without introducing subjectivity. Another potential threat arises from the manual labeling process used to classify the root causes and fixes of exception handling bugs. To reduce subjectivity, eight researchers with more than four years of software development experience independently conducted the labeling process following an open-coding approach commonly used in empirical studies [62–65]. The researchers received training and, during the analysis, collaboratively refined the categories to ensure consistency.

6.3. External validity

Although our dataset includes many popular Python projects across multiple domains, the results may not generalize to other programming languages or projects with different characteristics. To mitigate this threat, we selected projects from diverse domains within the Python ecosystem, reducing the likelihood that the identified taxonomy is domain-specific. Nevertheless, additional studies involving under-represented domains or proprietary industrial systems could further strengthen the external validity of our findings.

7. Related work

In this section, we present related work on exception handling bugs, exception handling practices, developers' perceptions, the evolution of exception handling bugs, and automated approaches.

7.1. Exception handling bugs

Ebert et al. [13] investigated the characteristics of exception handling bugs in Java through a survey with 154 developers and a manual classification of bugs by frequency, severity, and fixing difficulty, highlighting their prevalence and potential impact. They later reflected on how their work shaped subsequent research in the area [14]. Barbosa et al. [15] analyzed exceptional faults in Tomcat and Hadoop and identified macro-categories of exception-related faults (e.g., suppressed exceptions, destructive remapping, overly protective try-blocks), emphasizing the role of insufficient information and inappropriate handling.

Complementary to manual categorizations, da Silva et al. [66] proposed an approach to automatically label exception handling bugs from 10 years of bug-fixing activity in Apache Hadoop, reducing the manual effort required to curate exception-related bug datasets. Pádua et al. [6] examined how exception handling practices relate to post-release defects in Java/C# projects, reporting fewer defects in projects with better practices; rather than correlating practices with defect rates, our study focuses on characterizing Python exception handling bugs through root causes, fix patterns, and the presence of exception handling anti-patternsduring bug fixing.

Prior work also studied context-specific ecosystems. Lo et al. [29] analyzed exception handling bugs and fixes in Android apps, showing that many bugs are related to platform APIs and runtime exceptions. Chen et al. [44] examined exception bugs in large-scale cloud systems and reported frequent triggering by non-semantic conditions, with many issues related to network and file system errors; they also proposed a detection tool (DIET). Sawadpong et al. [67] linked exception handling mechanisms to increased defect density in Eclipse releases, motivating deeper analyses of exception-related defects.

Although not centered on exception handling bugs, Khan et al. [50] assessed the preventive value of static type checking in Python by retrospectively applying PEP-484 annotations and *mypy* to fixed defects in 210 projects, finding that a portion of defects could have been prevented and identifying common anti-patterns (e.g., reference redefinition, dynamic attribute initialization, null handling). These results reinforce that Python-specific language characteristics can systematically contribute to runtime failures that often surface as exceptions.

Finally, Coelho et al. [43] analyzed thousands of Android exception stack traces and reported that many failures stem from logic errors and resource constraints, and that a survey indicated limited awareness of undocumented exception behavior. Hassan et al. [68] studied exceptions in the field via search logs, showing their practical impact and recurrence across projects.

Overall, the closest work to ours remains Java-centered empirical studies of exception handling bugs [13,15]. Our study differs by (i) providing a large-scale characterization of exception handling bugs in Python projects, (ii) jointly analyzing root causes and fix patterns (rather than only categorizing bugs), and (iii) empirically examining exception handling anti-patternsappearing during bug fixing, offering evidence that can inform future detection/repair tools and developer guidelines.

7.2. Exception handling practices

Several studies have cataloged exception handling smells and refactorings. Rocha et al. [22] proposed a catalog of bad practices and refactoring actions, grounded in the capabilities of static analysis tools,

while Tarwani and Chug [69] identified additional smell types and proposed a template to describe them systematically.

Large-scale analyses further examined practices in real codebases. Pádua et al. [9] analyzed exception handling blocks and exception flows in Java and C# libraries, reporting frequent cross-method exception flows and limited documentation. Kim et al. [10] studied Java libraries and bug reports, highlighting poor documentation of runtime exceptions and a substantial presence of anti-patterns, suggesting that developers may underestimate the impact of poor practices. From a validation perspective, Hora et al. [70] reported that exceptional behaviors are under-tested compared to normal flows, indicating a gap between implementation and testing of exception handling.

Developer- and process-oriented studies provide complementary context. Melo and Treude [71] derived exception handling guidelines from interviews and surveys, showing that guidelines often exist and are disseminated through reviews. Jalote et al. [72] investigated robustness in the presence of exception handling changes, indicating fragility and opportunities for improvement. João et al. [73] analyzed exception handling-related pull requests and found that many changes target robustness improvements rather than bug fixes, frequently emphasizing user-facing error representation and practical recovery actions. Kechagia et al. [11] studied exceptions in Android and observed that undocumented exceptions contribute to failures, although documentation alone may not change handling behavior.

In contrast to prior work focusing on Java/C# practices, policies, and documentation [9–11], our study examines how such practices manifest in Python specifically during the fixing process of exception handling bugs, connecting bugs, fixes, and exception handling anti-patternsin a unified empirical characterization.

7.3. Exception handling evolution

Research on evolution has examined how exception handling smells and constructs change over time. Pádua and Shang et al. [21] studied the prevalence of exception smells in Java/C# projects and showed that while many smell types occur, a small subset is particularly prevalent and often tied to complex exception flows. Osman et al. [7,74] investigated long-lived Java systems and reported relatively stable ratios of exception handling constructs over time, with project type and domain influencing adoption patterns (e.g., applications vs. libraries). Cacho et al. [12] related changes in exception code to robustness metrics and found that exception handling often requires substantial modifications over time. Sousa et al. [8,20] combined surveys and release-based analysis in a large Java web system, suggesting that the lack of policy and documentation contributes to the continued introduction of smells.

While these studies provide important evidence on the prevalence and evolution of exception handling practices in Java/C#, our work instead focuses on Python exception handling bugsand complements evolution research by empirically examining bad practices that arise specifically during bug fixes.

7.4. Developers' perceptions (developer-centric studies)

Developer-centric work highlights why exception handling remains challenging in practice. Cabral and Marques [75] observed that many handlers focus on basic actions such as logging, re-throwing, or returning. Shah et al. [76,77] reported that developers often dislike exception handling structures, that novices may neglect exception handling due to perceived complexity, whereas experts treat it as essential, and they proposed models of exception handling strategies grounded in developer expertise and fault scope. Kery et al. [40] analyzed millions of try/catch blocks and found that local handling is common and that poor practices (e.g., empty handlers and generic exceptions) remain prevalent. Asaduzzaman et al. [38] likewise identified improper practices at scale and discussed how developers construct exception chains and messages.

Building on these insights, our study investigates how Python developers address exception handling bugsin practice and shows that exception handling anti-patternscan still appear even during bug fixing, motivating guidance and tool support tailored to Python.

7.5. Automated approaches in exception handling

Automated approaches address complementary exception handling tasks, including mining missing API error-handling specifications [51], detecting inconsistencies between thrown exceptions and messages [47], recommending exception types [49], and suggesting exception handling strategies [78]. Other studies focus on where and how exceptions should be handled. For example, Jia et al. [52] proposed EHAdvisor to recommend handling locations, while Nguyen et al. [54] proposed FuzzyCatch to predict potential exceptions and recommend handling code. More recent work has explored learning-based and neural approaches, including exception handler generation [53], a CodeBERT-based programming assistant [55], and neural exception handling recommendation [56].

Recent studies also investigate LLM-based support for exception handling. For instance, *Seeker* uses a multi-agent LLM-based framework to detect fragile code, identify exception types, and generate handling solutions [57]. Similarly, SHIELDA proposes structured handling of exceptions in LLM-driven agentic workflows by defining a taxonomy of agentic workflow exceptions and organizing recovery through classification, handling patterns, flow control, and state recovery [58]. Our study complements these automated efforts by providing empirical evidence of Python-specific root causes, fix patterns, and exception handling anti-patterns observed in real bug-fixing activities. This evidence can inform future tools that detect Python exception handling bugs, recommend context-aware repairs, and validate whether generated handlers introduce anti-patterns.

8. Conclusion

This paper presents an exploratory study of exception handling bugs in Python, analyzing their root causes, fix patterns, and their relation to exception handling anti-patterns. From 942 exception handling bugs mined across 550 open-source Python projects, we identified 12 distinct root causes and 25 fix patterns. *Unhandled Exceptions* account for more than 50% of the root causes, while *Exception Handling Block Operations* and *Improve Raising Conditions* together represent almost 57% of applied fixes, revealing concentrated, structured repair strategies. We also observed that many exception handling anti-patterns were already present before the fix and preserved after the repair. However, exception handling fixes still resulted in a minor increase in anti-pattern occurrences, highlighting that maintenance activities may introduce or perpetuate poor exception handling practices. In future studies, we plan to evaluate LLM-generated repairs for Python exception handling bugs using our dataset and taxonomy as benchmarks for exception handling repair. In particular, we aim to assess whether LLMs can identify missing exception handling logic, recommend suitable exception types, generate handlers consistent with the surrounding program context, and avoid introducing exception handling anti-patterns.

CRediT authorship contribution statement

Jairo Souza: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Data curation, Conceptualization. **Eric Coelho:** Validation, Software, Resources, Methodology. **João Correia:** Validation, Resources. **Rodrigo Lima:** Writing – review & editing, Methodology, Investigation. **Leopoldo Teixeira:** Writing – review & editing, Supervision, Resources, Project administration, Conceptualization. **Baldoino Fonseca:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. Leopoldo and Baldoino are partially supported by CNPq grants 310405/2025-4 and 309227/2025-9.

Data availability

The link to the data is shared in the paper.

References

[1] J. Bloch, Effective java™, second edition, second ed., Prentice Hall Press, USA, 2008.
[2] S. Luo, A. Sheth, K. Kochut, J. Miller, Exception handling in workflow systems, Appl. Intell. 13 (2002) http://dx.doi.org/10.1023/A:1008388412284.
[3] P. Buhr, W. Mok, Advanced exception handling mechanisms, IEEE Trans. Softw. Eng. 26 (9) (2000) 820–836, http://dx.doi.org/10.1109/32.877844.
[4] Oracle, What is an exception? 2014, URL: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html,
[5] S. Nakov, V. Kolev, Fundamentals of Computer Programming with C#, Faber Publishing, 2013-09-01.
[6] G. B. de Pádua, W. Shang, Studying the relationship between exception handling practices and post-release defects, in: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories, MSR, 2018, pp. 564–575.
[7] H. Osman, A. Chiș, C. Corrodi, M. Ghafari, O. Nierstrasz, Exception Evolution in Long-lived Java Systems, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR, 2017, pp. 302–311, http://dx.doi.org/10.1109/msr.2017.21.
[8] D.B.C.d. Sousa, P.H.M. Maia, L.S. Rocha, W. Viana, Studying the evolution of exception handling anti-patterns in a long-lived large-scale project, J. Braz. Comput. Soc. 26 (1) (2020) 1, http://dx.doi.org/10.1186/s13173-019-0095-5.
[9] G.B.D. Pádua, W. Shang, Revisiting Exception Handling Practices with Exception Flow Analysis, in: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation, SCAM, 2017, pp. 11–20, http://dx.doi.org/10.1109/scam.2017.16.
[10] M. Kim, R. Robbes, C. Bird, D. Sena, R. Coelho, U. Kulesza, R. Bonifácio, Understanding the exception handling strategies of Java libraries, Proc. the 13th Int. Conf. Min. Softw. Repos. (2016) 212–222, http://dx.doi.org/10.1145/2901739.2901757.
[11] M. Kechagia, M. Fragkoulis, P. Louridas, D. Spinellis, The exception handling riddle: An empirical study on the Android API, J. Syst. Softw. 142 (2018) 248–270, http://dx.doi.org/10.1016/j.jss.2018.04.034.
[12] N. Cacho, E.A. Barbosa, J. Araujo, F. Pranto, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, I. Garcia, How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 31–40, http://dx.doi.org/10.1109/icsme.2014.25.
[13] F. Ebert, F. Castor, A. Serebrenik, An exploratory study on exception handling bugs in Java programs, J. Syst. Softw. 106 (2015) 82–101, http://dx.doi.org/10.1016/j.jss.2015.04.066.
[14] F. Ebert, F. Castor, A. Serebrenik, A reflection on "an exploratory study on exception handling bugs in java programs", in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, SANER, 2020, pp. 552–556, http://dx.doi.org/10.1109/SANER48275.2020.9054791.
[15] E.A. Barbosa, A. Garcia, S.D.J. Barbosa, Categorizing faults in exception handling: A study of open source projects, in: 2014 Brazilian Symposium on Software Engineering, 2014, pp. 11–20, http://dx.doi.org/10.1109/SBES.2014.19.
[16] Y. Peng, Y. Zhang, M. Hu, An Empirical Study for Common Language Features Used in Python Projects, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, 00 (2021) 24–35, http://dx.doi.org/10.1109/saner50967.2021.00012.
[17] ISO/IEC/IEEE international standard - systems and software engineering–Vocabulary, ISO/IEC/IEEE 24765:2017(E) (2017) 1–541, http://dx.doi.org/10.1109/IEEESTD.2017.8016712.
[18] D. Beazley, Python Distilled, Addison-Wesley Professional, 2021.
[19] P. Laplante, C. Neill, Antipatterns: Identification, refactoring, and management, 2005, pp. 1–289, http://dx.doi.org/10.1201/9781420031249,
[20] D.B.C.d. Sousa, P.H. Maia, L.S. Rocha, W. Viana, Analysing the Evolution of Exception Handling Anti-Patterns in Large-Scale Projects: A Case Study, Proc. the VII Braz. Symp. Softw. Components, Archit. Reuse - SBCARS '18 (2018) 73–82, http://dx.doi.org/10.1145/3267183.3267191.
[21] G.B.d. Pádua, W. Shang, Studying the Prevalence of Exception Handling Anti-Patterns, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension, ICPC, 2017, pp. 328–331, http://dx.doi.org/10.1109/icpc.2017.1.
[22] J. Rocha, H. Melo, R. Coelho, B. Sena, Towards a catalogue of java exception handling bad smells and refactorings, in: Proceedings of the 25th Conference on Pattern Languages of Programs, PLoP '18, The Hillside Group, USA, 2018.
[23] C.Y. Hsieh, C.L. My, K.T. Ho, Y.C. Cheng, Identification and refactoring of exception handling code smells in JavaScript, J. Internet Technol. 18 (6) (2017) 1461–1471, URL: https://jit.ndhu.edu.tw/article/view/1597.
[24] B.L. Berg, H. Lune, Qualitative research methods for the social sciences, eighth ed., Pearson, Upper Saddle River, NJ, 2011.
[25] O. Dabic, E. Aghajani, G. Bavota, Sampling projects in GitHub for MSR studies, in: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, IEEE, 2021, pp. 560–564.
[26] J. Souza, Complementary material, 2024, URL: https://github.com/exception-handling/exception-handling-bugs.
[27] J. Souza, T. Alves, R. Oliveira, L. Teixeira, B. Fonseca, Exception miner: Multi-language static analysis tool to identify exception handling anti-patterns, in: Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software, SBC, Porto Alegre, RS, Brasil, 2024, pp. 741–747, http://dx.doi.org/10.5753/sbes.2024.3573, URL: https://sol.sbc.org.br/index.php/sbes/article/view/30420.
[28] M. Brunsfeld, Tree-sitter-a new parsing system for programming tools, in: Strange Loop Conference, 2018, https://www.thestrangeloop.com//tree-sitter—a-new-parsing-system-for-programming-tools.html.
[29] D. Lo, D. Kim, E. Gamess, T.T. Nguyen, P.M. Vu, T.T. Nguyen, An Empirical Study of Exception Handling Bugs and Fixes, Proc. the 2019 ACM Southeast Conf. ZZZ - ACM SE '19 (2019) 257–260, http://dx.doi.org/10.1145/3299815.3314472.
[30] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: How misclassification impacts bug prediction, in: 2013 35th International Conference on Software Engineering, ICSE, 2013, pp. 392–401, http://dx.doi.org/10.1109/ICSE.2013.6606585.
[31] X.-B.D. Le, L. Bao, D. Lo, X. Xia, S. Li, C. Pasareanu, On reliability of patch correctness assessment, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, 2019, pp. 524–535, http://dx.doi.org/10.1109/ICSE.2019.00064.
[32] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, USA, 2008.
[33] J. Cohen, A coefficient of agreement for nominal scales, Educ. Psychol. Meas. 20 (1960) 37–46, URL: https://api.semanticscholar.org/CorpusID:15926286.
[34] k. krippendorff, Systematic and random disagreement and the reliability of nominal data, Dep. Pap. (ASC) 2 (2008) http://dx.doi.org/10.1080/19312450802467134.
[35] X.B.D. Le, L. Bao, D. Lo, X. Xia, S. Li, C. Pasareanu, On reliability of patch correctness assessment, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, 2019, pp. 524–535, http://dx.doi.org/10.1109/ICSE.2019.00064.
[36] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008, URL: http://www-csli.stanford.edu/~hinrich/information-retrieval-book.html.
[37] F. Falcão, C. Barbosa, B. Fonseca, A. Garcia, M. Ribeiro, R. Gheyi, On Relating Technical, Social Factors, and the Introduction of Bugs, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, SANER, 2020, pp. 378–388, http://dx.doi.org/10.1109/saner48275.2020.9054824, arXiv:1811.01918.
[38] M. Asaduzzaman, M. Ahasanuzzaman, C.K. Roy, K.A. Schneider, How developers use exception handling in java? in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 516–519, http://dx.doi.org/10.1145/2901739.2903500.
[39] R. Python, The most diabolical python antipattern – real python — realpython.com, 2025, https://realpython.com/the-most-diabolical-python-antipattern/. (Accessed 15 January 2025).
[40] M.B. Kery, C. Le Goues, B.A. Myers, Examining programmer practices for locally handling exceptions, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 484–487, http://dx.doi.org/10.1145/2901739.2903497.
[41] L.P. Ramos, LBYL vs EAFP: Preventing or handling errors in python, 2022, URL: https://realpython.com/python-lbyl-vs-eafp/,
[42] L.P. Ramos, Python2019s raise: Effectively raising exceptions in your code, 2024, URL: https://realpython.com/python-raise-exception/,
[43] R. Coelho, L. Almeida, G. Gousios, A.v. Deursen, C. Treude, Exception handling bug hazards in Android, Empir. Softw. Eng. 22 (3) (2017) 1264–1304, http://dx.doi.org/10.1007/s10664-016-9443-7.

[44] H. Chen, W. Dou, Y. Jiang, F. Qin, Understanding exception-related bugs in large-scale cloud systems, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2019, pp. 339–351, http://dx.doi.org/10.1109/ASE.2019.00040.

[45] L.P. Lima, L.S. Rocha, C.I.M. Bezerra, M. Paixao, Assessing exception handling testing practices in open-source libraries, Empir. Softw. Engg. 26 (5) (2021) http://dx.doi.org/10.1007/s10664-021-09983-3.

[46] A.J.A. da Silva, R.G. Vieira, D.P.P. Mesquita, J.P.P. Gomes, L.S. Rocha, Towards automatic labeling of exception handling bugs: A case study of 10 years bug-fixing in apache hadoop, Empir. Softw. Engg. 29 (4) (2024) http://dx.doi.org/10.1007/s10664-024-10494-0.

[47] L. Xu, H. Zhong, Detecting inconsistent thrown exceptions, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension, ICPC, 2021, pp. 391–395, http://dx.doi.org/10.1109/ICPC52881.2021.00045.

[48] H. Liu, S. Li, Z. Jia, Y. Zhang, L. Bai, S. Zheng, X. Mao, X. Liao, Error delayed is not error handled: Understanding and fixing propagated error-handling bugs, Proc. ACM Softw. Eng. 2 (FSE) (2025) http://dx.doi.org/10.1145/3729384.

[49] H. Zhong, Which exception shall we throw? in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22, Association for Computing Machinery, New York, NY, USA, 2023, http://dx.doi.org/10.1145/3551349.3556895.

[50] F. Khan, B. Chen, D. Varro, S. McIntosh, An empirical study of type-related defects in python projects, IEEE Trans. Softw. Eng. 48 (8) (2022) 3145–3158, http://dx.doi.org/10.1109/TSE.2021.3082068.

[51] M. Acharya, T. Xie, Mining API error-handling specifications from source code, in: Proceedings of the 12th International Conference on Fundamental Approaches To Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, FASE '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 370–384.

[52] X. Jia, S. Chen, X. Zhou, X. Li, R. Yu, X. Chen, J. Xuan, Where to Handle an Exception? Recommending Exception Handling Locations from a Global Perspective, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension, ICPC, 00 (2021) 369–380, http://dx.doi.org/10.1109/icpc52881.2021.00042.

[53] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, X. Liu, Learning to handle exceptions, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, Association for Computing Machinery, New York, NY, USA, 2021, pp. 29–41, http://dx.doi.org/10.1145/3324884.3416568.

[54] T. Nguyen, P. Vu, T. Nguyen, Code recommendation for exception handling, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1027–1038, http://dx.doi.org/10.1145/3368089.3409690.

[55] Y. Cai, A. Yadavally, A. Mishra, G. Montejo, T. Nguyen, Programming assistant for exception handling with codebert, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024, http://dx.doi.org/10.1145/3597503.3639188.

[56] Y. Li, T.N. Nguyen, Y. Cai, A. Yadavally, A. Mishra, G. Montejo, Neural exception handling recommender, in: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, in: ICSE-Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 316–317, http://dx.doi.org/10.1145/3639478.3643082.

[57] X. Zhang, Y. Chen, Y. Yuan, M. Huang, Towards exception safety code generation with intermediate representation agents framework, 2025, URL: https://arxiv.org/abs/2410.06949 arXiv:2410.06949.

[58] J. Zhou, J. Chen, Q. Lu, D. Zhao, L. Zhu, SHIELDA: Structured handling of exceptions in LLM-driven agentic workflows, 2025, URL: https://arxiv.org/abs/2508.07935 arXiv:2508.07935.

[59] C. Wohlin, P. Runeson, M. Hst, M.C. Ohlsson, B. Regnell, A. Wessln, Experimentation in Software Engineering, Springer Publishing Company, Incorporated, 2012.

[60] M.E. Haug, Fast Typesetting with Incremental Compilation (Ph.D. thesis), tu-berlin, 2022.

[61] S. Monnier, SMIE: Weakness is power!: Auto-indentation with incomplete information, 2020, arXiv preprint arXiv:2006.03103.

[62] M. Bagherzadeh, N. Fireman, A. Shawesh, R. Khatchadourian, Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences, Proc. the ACM Program. Lang. 4 (OOPSLA) (2020) 1–32, http://dx.doi.org/10.1145/3428282.

[63] D. Liu, Y. Feng, Y. Yan, B. Xu, Towards understanding bugs in Python interpreters, Empir. Softw. Eng. 28 (1) (2023) 19, http://dx.doi.org/10.1007/s10664-022-10239-x.

[64] J. Chen, Y. Liang, Q. Shen, J. Jiang, S. Li, Toward Understanding Deep Learning Framework Bugs, ACM Trans. Softw. Eng. Methodol. (2023) http://dx.doi.org/10.1145/3587155.

[65] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, X. Liu, Characterizing and Detecting WebAssembly Runtime Bugs, ACM Trans. Softw. Eng. Methodol. (2023) http://dx.doi.org/10.1145/3624743.

[66] A.J.A. da Silva, R.G. Vieira, D.P.P. Mesquita, J.P.P. Gomes, L.S. Rocha, Towards automatic labeling of exception handling bugs: A case study of 10 years bug-fixing in apache hadoop, Empir. Softw. Engg. 29 (4) (2024) http://dx.doi.org/10.1007/s10664-024-10494-0.

[67] P. Sawadpong, E.B. Allen, B.J. Williams, Exception Handling Defects: An Empirical Study, in: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, 2012, pp. 90–97, http://dx.doi.org/10.1109/hase.2012.24.

[68] F. Hassan, C. Bansal, N. Nagappan, T. Zimmermann, A.H. Awadallah, An empirical study of software exceptions in the field using search logs, in: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3382494.3410692.

[69] S. Tarwani, A. Chug, Illustration and Detection of Exception Handling Bad Smells, in: 2021 8th International Conference on Computing for Sustainable Global Development (INDIACom), 2021, pp. 804–810, http://dx.doi.org/10.1109/indiacom51348.2021.00144.

[70] A. Hora, G. Fraser, Exceptional behaviors: How frequently are they tested? in: 2025 IEEE/ACM International Conference on Automation of Software Test, AST, 2025, pp. 70–79, http://dx.doi.org/10.1109/AST66626.2025.00014.

[71] H. Melo, R. Coelho, C. Treude, Unveiling Exception Handling Guidelines Adopted by Java Developers, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, SANER, 2019, pp. 128–139, http://dx.doi.org/10.1109/saner.2019.8668001.

[72] P. Jalote, L. Briand, A.v.d. Hoek, N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, E.A. Barbosa, A. Garcia, Trading robustness for maintainability: an empirical study of evolving c# programs, Proc. the 36th Int. Conf. Softw. Eng. (2014) 584–595, http://dx.doi.org/10.1145/2568225.2568308.

[73] J. Correia, D. Coutinho, A. Garcia, R. de Mello, C. Barbosa, A. Oliveira, W.K.G. Assunção, J.A. Pereira, I. Steinmacher, M. Gerosa, J. Souza, J. Arriel, On the investigation of exception pull request characteristics: Exploring the apache ecosystem, in: 2024 IEEE International Conference on Source Code Analysis and Manipulation, SCAM, 2024, pp. 143–153, http://dx.doi.org/10.1109/SCAM63643.2024.00023.

[74] H. Osman, A. Chiş, J. Schaerer, M. Ghafari, O. Nierstrasz, On the Evolution of Exception Usage in Java Projects, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, 2017, pp. 422–426, http://dx.doi.org/10.1109/saner.2017.7884646.

[75] B. Cabral, P. Marques, Exception handling: A field study in java and .net, in: E. Ernst (Ed.), ECOOP 2007 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 151–175.

[76] H. Shah, C. Görg, M.J. Harrold, Why do developers neglect exception handling? in: Proceedings of the 4th International Workshop on Exception Handling, WEH '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 62–68, http://dx.doi.org/10.1145/1454268.1454277.

[77] H. Shah, C. Gorg, M. Harrold, Understanding Exception Handling: Viewpoints of Novices and Experts, IEEE Trans. Softw. Eng. 36 (2) (2010) 150–161, http://dx.doi.org/10.1109/tse.2010.7.

[78] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, J. Xuan, EH-Recommender: Recommending Exception Handling Strategies Based on Program Context, in: 2018 23rd International Conference on Engineering of Complex Computer Systems, ICECCS, 2018, pp. 104–114, http://dx.doi.org/10.1109/iceccs2018.2018.00019.