Towards Understanding Performance Bugs in Popular Data Science Libraries

HAOWEN YANG, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
ZHENGDA LI, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
ZHIQING ZHONG, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
XIAOYING TANG, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
PINJIA HE*, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

With the increasing demand for handling large-scale and complex data, data science (DS) applications often suffer from long execution time and rapid RAM exhaustion, which leads to many serious issues like unbearable delays and crashes in financial transactions. As popular DS libraries are frequently used in these applications, their performance bugs (PBs) are a major contributing factor to these issues, making it crucial to address them for improving overall application performance. However, PBs in popular DS libraries remain largely unexplored. To address this gap, we conducted a study of 202 PBs collected from seven popular DS libraries. Our study examined the impact of PBs and proposed a taxonomy for common root causes. We found over half of the PBs arise from inefficient data processing operations, especially within data structure. We also explored the effort required to locate their root causes and fix these bugs, along with the challenges involved. Notably, around 20% of the PBs could be fixed using simple strategies (e.g. Conditions Optimizing), suggesting the potential for automated repair approaches. Our findings highlight the severity of PBs in core DS libraries and offer insights for developing high-performance libraries and detecting PBs. Furthermore, we derived test rules from our identified root causes, identifying eight PBs, of which four were confirmed, demonstrating the practical utility of our findings.

CCS Concepts: • **Software and its engineering → Software performance**.

Additional Key Words and Phrases: empirical study, data science, performance problem

ACM Reference Format:
Haowen Yang, Zhengda Li, Zhiqing Zhong, Xiaoying Tang, and Pinjia He. 2025. Towards Understanding Performance Bugs in Popular Data Science Libraries. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE104 (July 2025), 24 pages. https://doi.org/10.1145/3729374

1 Introduction

As data volume and complexity grow, performance bugs (PB) in DS applications increasingly lead to long execution time and excessive memory consumption [76]. Such performance issues not only degrade user experience but also cause significant financial losses for businesses [24, 27]. Popular

*Corresponding author.

Authors' Contact Information: Haowen Yang, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), Shenzhen, China, haowenyang1@link.cuhk.edu.cn; Zhengda Li, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), Shenzhen, China, zhengdali1@link.cuhk.edu.cn; Zhiqing Zhong, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), Shenzhen, China, zhiqingzhong@link.cuhk.edu.cn; Xiaoying Tang, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), Shenzhen, China, tangxiaoying@cuhk.edu.cn; Pinjia He, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), Shenzhen, China, hepinjia@cuhk.edu.cn.

ACM 2994-970X/2025/7-ARTFSE104
https://doi.org/10.1145/3729374

DS libraries like *Pandas* and *NumPy*, which provide basic functions for various DS tasks, play a critical role in these applications [29, 76]. The importance of analyzing performance-related bugs in these libraries has been increasingly recognized [29], as issues like memory leaks can lead to system crashes and financial losses in dependent applications. For instance, due to the PB in the *Pandas* function `rolling().max()`, a popular open-source cryptocurrency trading bot with 24.3k stars in Github, *Freqtrade* [2], has a memory leak issue. This issue [3] impacted many developers using *Freqtrade* for quantitative trading, leading to application crashes and significant financial losses. One of them expressed their frustration: '`I was desperate not being able to use my strategy that works quite well.`'

Such incidents highlight that PBs are not just minor inconveniences but represent a significant risk to DS applications, particularly in high-stakes environments like finance. The ongoing discussions within the DS community underscore the prevalence of these issues. As of November 2023, there are 1,449 Stack Overflow questions tagged with both *Pandas* and *Performance*, attracting over one million views, with a 58% answer acceptance rate. On average, it takes 31 days for accepted answers to be proposed. On GitHub, *Pandas* has 911 performance-labeled issues. Although 85% are closed, they take a considerable time to resolve, with a median resolution time of 99 days.

Despite the severity and prevalence of PBs in DS, there is not enough research on them. A lot of effort has been made to study the characteristics of general [29, 34, 42] and specific functional bugs [36, 52, 65] encountered by DS practitioners. In addition, some research highlights developers' limited understanding of API optimal performance, which causes inefficient choices and, ultimately, PBs in DS apps [76, 77]. However, these studies are not specifically designed for PBs or do not target popular DS libraries, thus only capturing partial characteristics of PBs in DS libraries. In comparison, PBs have been widely studied for other domains, e.g., mobile applications [40, 41, 56, 57], desktop applications [39, 44, 46, 55, 59, 62, 63, 74, 83, 85, 86], embedded systems [73], configurable systems [49, 51], database-backed web applications [81, 82], JavaScript systems [70] and deep learning systems [35, 67, 78]. However, the characteristics of PBs in these fields differ from those in popular DS libraries due to DS specific knowledge. Specifically, PBs in DS libraries often stem from improper data structure management (Sec. 3.2), such as inefficient index retrieval or incomplete optimizations due to limited understanding of newly introduced data structures. These issues rarely appear in general PBs [55, 85]. Additionally, a common challenge in DS libraries involves substantial code changes when replacing data structures (Sec. 3.3), which is a phenomenon much less frequent in other domains. To sum up, there is a limited understanding of the characteristics of PBs in these crucial libraries.

To bridge this knowledge gap, we present the first in-depth study to characterize PBs in DS libraries. The study aims to answer the following four research questions:

- **RQ1 Impact Scope and Symptoms:** What is the impact scope of PBs in popular DS libraries on dependent repositories? What are the symptoms of these PBs?
- **RQ2 Root Causes:** What are the root causes of these PBs?
- **RQ3 Effort and Challenges:** What are the efforts involved in locating PBs' root causes and fixing them? Are some of the PBs difficult to diagnose and fix? If yes, what are the challenges.
- **RQ4 Common Fixing Strategy:** For those PBs with small fixing efforts, what are their common fixing strategies?

We collect 202 PBs from seven popular DS libraries (Table 1), and manually investigate the PBs to characterize their impacts (**RQ1**) and root causes (**RQ2**). Next, we examine the effort required to locate PBs' root causes and fix them, and then highlight the key challenges (**RQ3**). Finally, we provide common fixing strategies for the PBs with small fixing efforts (**RQ4**).

Table 1. Statistics of Popular DS Libraries

Library	Category	# Stars	# Forks	# GitHub issues
Requests	Data Acquisition	51.9k	9.3k	3,959
Numpy	Scientific Computing	27.3k	9.7k	12,681
SciPy	Scientific Computing	12.8k	5.1k	10,217
Pandas	Data Analysis	43.0k	17.7k	26,710
Scikit-learn	Data Analysis	59.2k	25.2k	11,082
Matplotlib	Visualization	19.8k	7.5k	1,0432
Seaborn	Visualization	12.3k	1.9k	2,540

Through investigating the four RQs, we have discovered that PBs in DS core libraries have a widespread and significant impact on the entire DS ecosystem. For instance, our investigation into the explicit spread of PBs shows that over 70% of affected repositories have more than 100 stars, and more than 40% feature over 10,000 stars. Many performance issues in downstream libraries within the DS ecosystem remain unidentified. Some of them cannot be resolved by simply updating the versions of upstream libraries.

As for root causes, we found that over half of the PBs arise from *Inefficient Data Manipulation Operations*, especially within data structures, while *Memory Management*, *Scientific Computing*, and *I/O* issues account for more than a third of these problems. Furthermore, we have identified three challenges in locating and four challenges in fixing these PBs. These challenges primarily stem from understanding specific data structures and the extensive refactoring required when modifying them.

Our findings also reveal actions that researchers and core DS library developers can take to improve performance across the DS ecosystem. This includes detecting, locating and fixing PBs in popular DS libraries, as well as monitoring and preventing the spread of PBs throughout the ecosystem. For example, with the identified root causes, one could develop test oracles or propose guidelines to avoid certain PBs during development (detailed further in Section 4).

To demonstrate the usefulness of our findings, we selected a few patterns from our bug taxonomy to create test oracles for further testing within the studied libraries. We tested eight cases and opened issues to report our findings to the developers. Four of these were confirmed as PBs requiring fixes.

In summary, the paper makes the following contributions.

- We conduct the first in-depth study to characterize 202 PBs in seven popular DS libraries.
- We provide a taxonomy of root causes and symptoms of PBs in popular DS libraries.
- Our findings highlight the challenges of locating and fixing performance bugs in DS libraries, as well as common low-effort fixing strategies. These insights can serve as the foundation for designing future automation tools.
- We construct a dataset of PBs in these popular DS libraries. The dataset can be further used in future research on detecting and fixing PBs.

2 Study Design

This section provides an overview of our study. First, we explain how we selected DS libraries (Sec. 2.1) for analysis. Next, we describe how we collected PB-related issues and corresponding PRs from the selected DS libraries, and PB identification process (Sec. 2.2) as outlined in Fig. 1 (steps 1 and 2). Finally, in Sec. 2.3, we provide a detailed analysis of PB characteristics. This analysis includes aspects such as impact scope, symptoms, root causes, and fixing strategies. As shown in Fig. 1, these elements correspond to steps 3, 4, 5, and 6.

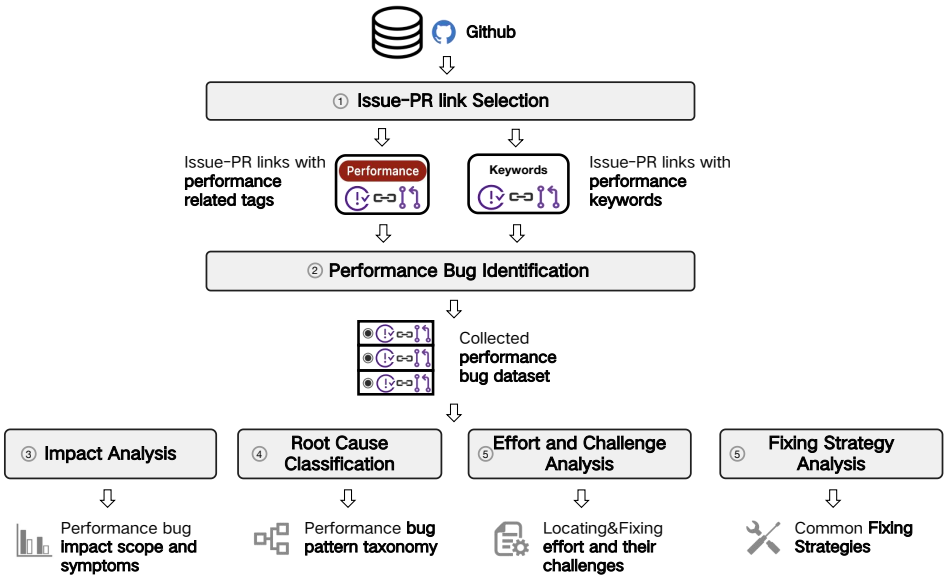


Fig. 1. Overview of our study design.

2.1 DS Libraries Selection

To enhance the practicality of our study, we simplified the 11 stages of DS pipeline in previous work [34] into four key stages in Fig. 2: *Data Acquisition*, *Scientific Computing*, *Data Analysis*, and *Visualization*. This simplification was necessary as some original stages, like *evaluation* and *prediction*, were too detailed for relevant DS libraries, while others, such as *communication* and *deployment*, focused more on system maintenance than functionality.

Drawing on previous work [29, 76] and popular technical websites [1, 25, 28], we initially selected 11 libraries, commonly used and discussed by DS developers on Stack Overflow, for the four key stages. This ensures that our chosen libraries are widely recognized and utilized within the DS community. For these 11 candidates, we further selected those that satisfy the following three criteria. First, we only chose Python projects hosted on Github as the subjects of our study (traceability) [57]. Second, a candidate library should have achieved more than 10,000 stars and 1,000 forks (popularity). Third, it should have at least 1,000 issues in total (maintainability). For the *data acquisition* stage, *Requests* [26] stands out for their efficiency and ease of use in fetching online data. *Numpy* [50] and *Scipy* [79] are included in *Scientific Computing* for their extensive range of numerical computing. In *data analysis* stage, *Pandas* [61]and *Scikit-learn* [66] are chosen due to their powerful data operations, and machine learning algorithms, which are crucial for analyzing large datasets. For *data visualization*, *Matplotlib* [33] and *Seaborn* [4] are selected due to their flexibility and ability to produce a wide range of high-quality plots and graphs.

2.2 Data Collection

To investigate the characteristics of PBs in popular DS libraries, we systematically collected data on PB-related *closed* issues and corresponding *merged* pull requests (PRs), following existing work [37, 47, 54, 71], and then reviewed them to identify genuine PBs acknowledged by developers. We focus only on issues accepted and integrated into the code repository, as they reflect PBs deemed significant by developers [37, 70]. Studying such issue-PR pairs is crucial for gaining a comprehensive understanding of the PBs. The comments in issues, combined with the code

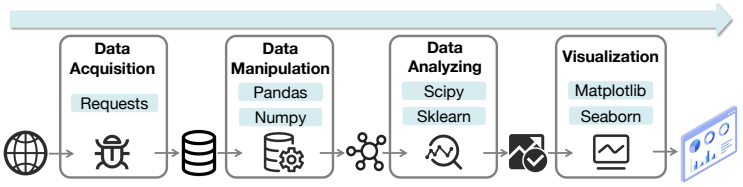


Fig. 2. Pipeline of DS Application

modifications in PRs, significantly facilitate the identification of the characteristics (e.g., common root causes). Additionally, the issues provide valuable data such as the number of comments, reflecting the efforts invested in non-code-related activities by developers when addressing PBs. The changes made in the code and the associated discussions within PRs are instrumental for understanding the root causes of PBs, effective solutions, and comprehending the rationale behind these solutions [37]. We detail the collecting operations below:

Step 1: Issue-PR Link Selection. Due to the time-consuming nature of manually analyzing PBs, collecting all PBs for manual inspection is impractical. Furthermore, as DS libraries evolve, PBs from the distant past may not provide relevant insights for the current characteristics. Therefore, we collected issue-PR links closed after 2020-08-17 to exclude possible obsolete usage or function discussions, following similar approaches in previous studies [35, 37, 43, 54, 71].

We also observed that some repositories consistently tag issues related to PBs with a *Performance* label, while others use it inconsistently or not at all. Consequently, following previous research [70], we combined label identification with keyword searches to select relevant issues. Among our selected repositories, *Pandas* and *Scikit-learn* have explicit performance labels. We therefore extract 148 issue-PR links directly related to PBs by checking for these labels. However, we found other libraries lacked a specific label that comprehensively represented all performance-related issues. We empirically checked the most similar labels (e.g. *Enhancement*) in other libraries and found it cannot accurately locate PBs. For example, *#18921* [7] in *Scipy* focuses on removing the `._is_array` attribute from sparse arrays, a change that is more about code organization and consistency rather than performance optimization. To address this, similar to prior PB studies [55, 64, 85], we adopted a keyword-based searching approach for the rest libraries. Specifically, we crawled all the issues and their corresponding PRs which contain performance-related keywords in their title or body. The keywords are: *performance, fast, slow, latency, efficient, optimize, profiling, overhead and timeit.* We retained 765 issue-PR links following this keyword-based filtration. After completing the first step, our collection expanded to 913 issues along with their corresponding PRs.

Step 2: PB Identification. We manually reviewed the 913 issues to identify PBs acknowledged by developers and verify if the proposed solutions effectively addressed them. This process filtered out noise, such as issues with relevant keywords that were not PB-related, including documentation updates, feature additions, and functional bugs. For example, issue `#22949` [10] in `Numpy` reported a missing link in the documentation (*doc/source/reference/c-api/array.rst*) where the word '*efficient*' appeared, yet this issue was unrelated to PBs. Two authors independently examined each issue and corresponding PR, with a third resolving any discrepancies to reach a consensus. To quantify the agreement between authors, we employed Cohen's Kappa [60] coefficient. The coefficient calculated was 0.813, indicating a strong agreement. Ultimately, we selected 202 PBs for detailed study, a scale comparable to previous study, including 98 PBs in JavaScript programs [70], 109 PBs in desktop or server applications [55], and 70 PBs in mobile applications [57].

2.3 Understanding and Labeling PB Properties

2.3.1 Step 3: Impact Analysis

To understand the impact of PBs in popular DS libraries on the ecosystem, we analyzed them from two perspectives: scope and symptoms.

Scope. The scope is studied from the perspective of dependent repositories, for which we differentiated between explicit impacts and implicit impacts. Explicit impacts refer to cases where downstream developers directly report performance issues, specifically referencing problems in the DS libraries. This helps us quantify how widely these issues affect dependent libraries. Implicit impacts refer to the potential performance vulnerabilities in dependent libraries that developers may unknowingly adopt. Investigating implicit impacts allows us to understand the hidden risks and the difficulty developers may face in detecting and addressing these issues.

To estimate explicit impacts, we developed a script that automates the extraction of references within issues. Specifically, the script accesses the collected PB related issues, parses the HTML content, and identifies links referencing other repositories by checking the `data-hovercard-url` attribute, which contains information about referenced repositories, issues, or users. If such references are found, the script checks whether the referred elements are issues and then extracts the repository names and corresponding issue numbers. We then manually reviewed the linked issues to confirm if they were PB-related and analyzed the symptoms and characteristics of the affected dependent libraries. Two authors independently reviewed all issue contents, including titles, descriptions, comments, and answers, to ensure the accuracy of our selection. A third author was consulted to resolve any differences in cases of disagreement.

To estimate implicit impacts, we first identify the APIs having PBs, then run performance tests across different library versions to determine the affected range. Next, we search GitHub for dependent libraries using the problematic version of the popular DS libraries by examining dependency files (e.g., *requirements.txt*) and select those with over 100 stars to ensure popularity. If a PB-related API is found in a dependent library, we manually check its usage in production code, as this directly impacts performance in real-world scenarios. If confirmed, we report the issue to alert developers of potential performance problems.

Symptoms. Furthermore, we labelled each PB's *symptom* based on its main consequence. We first randomly sample 50% of issues for pilot labeling. The first two authors manually analyzed the collected data and labeled each PB through an open-coding [69] procedure, following methodologies from existing studies [37, 54, 71]. Our analysis included a detailed examination of issues and PRs, focusing on titles, question descriptions, developer comments, code changes, as well as references to other issues or PRs.

2.3.2 Step 4: Root Cause Classification

We used open coding to identify preliminary tags for root causes, following previous work [85]. We derived these codes by extracting and summarizing relevant text on root causes from all available data. Starting with a random selection of 40 issues, two authors created an initial set of codes, which were then discussed and consolidated into a codebook. Any discrepancies were resolved with the help of a third author. For the remaining issues, we summarized the root causes briefly and categorized them using the established codebook. The authors hold group discussion for resolving uncertain cases in this process. The initial codebook covered most issues except for certain types like *Unnecessary Dependency Loading* and *Legacy Dependency Bottleneck*.

Subsequently, following previous work [32, 35], we assigned the labels based on the library's main functionalities to construct a taxonomy of root causes. We began by grouping tags corresponding to similar concepts into categories, with a focus on common DS specific misuse patterns. Parent

categories were then created based on the functional aspects to construct a taxonomy of root causes, such as *data manipulation* and *scientific computing*. This approach allows us to directly align our analysis with the library's core functionalities, making the findings more practical and relevant. The taxonomy underwent three iterations and each version was refined through discussions in physical meetings by all authors.

2.3.3 Step 5 & 6: Effort, Challenges and Common Fixing Strategy

To assess the effort required to locate PBs' root causes and fix them in popular DS libraries, we conducted a detailed analysis of the PBs. Following a previous work [57], we compare the root causes locating and bug-fixing effort with non-performance bugs from the repository of *Pandas* and *Scikit-learn* (we selected 100 bugs for each). In line with previous research [57], non-performance bugs from other repositories were excluded due to their limited PBs (less than 10), which could lead to inconclusive comparisons and weaken the foundation for further studies [45].

Challenging bugs typically remain open longer, generate more developer discussions, and require larger patches [57]. As a result, we measured root causes locating and bug-fixing efforts using the three metrics, consistent with methodologies used in existing studies [57, 83]: (1) *Bug Open Duration*: The time elapsed from a bug report is opened to when the bug is fixed; (2) *Number of Bug Comments*: This metric quantifies the discussions among developers and users during the bug's lifecycle; (3) *Patch Size*: Measured by the number of lines of code (LOC) changed to resolve the bug. To ensure patch size accurately reflects the effort required to fix performance bugs, we excluded non-production files (e.g., test, benchmark, documentation) and removed empty lines when extracting LOC. Additionally, during manual code review, we found some PRs delete and add it back a paragraph of code where some lines' content are not changed, they are simply deleted and added it back alongside required modifications. These changes were counted in the LOC but did not alter the content at all. We also excluded these from the PR LOC calculation. Furthermore, to understand the statistical significance of these differences, we employed the Mann-Whitney U-test (U-test) [58] to test three hypotheses:

- The time taken to locate root causes and fix PBs is not significant longer than non-performance issues.
- The amount of discussion for PBs is not significant greater than for non-performance issues.
- The patch sizes for PB fixes are not significant larger compared to non-performance bug fixes.

We used the non-parametric U-test due to the unknown data distribution, as it is more robust without relying on distributional assumptions. Additionally, we selected Cliff's Delta to measure effect size, independent of distribution.

To explore why locating PBs' root causes and fixing them in DS libraries is challenging, we examined issue discussions and related code changes. Building on previous work [80], we analyzed issues that were closed after one month with over 10 comments and manually reviewed the top 25% of PBs by LOC to understand the challenges and efforts required to resolve these bugs in popular DS libraries. For the common fixing strategy, we manually reviewed PBs with LOC not larger than 10 and assigned labels using a procedure similar to the root cause labeling process (2.3.2) described earlier.

3 Results

3.1 Impact Analysis (RQ1)

3.1.1 Scope Analysis

The impact scope of PBs in core DS libraries is illustrated from two perspectives: *explicit impacts*, which are estimated through directly reported PBs in issues, and *implicit impacts*, which are

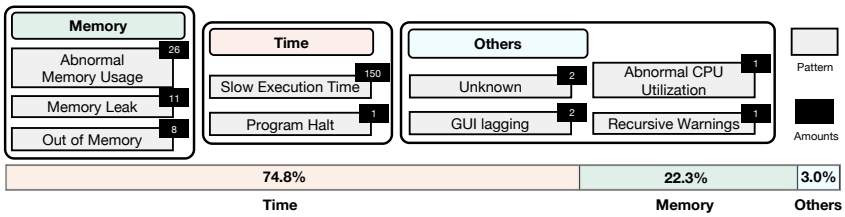


Fig. 3. Symptoms of PBs in DS Libraries.

estimated by potential performance vulnerabilities adopted without the awareness of developers from dependent repositories.

Explicit Impact. We found 41 issues directly linked to the PBs we identified. These issues affect a diverse range of dependent repositories, with stars varying from 2 to 26.4k. 78% of the impacted repositories are highly popular and widely used, each with more than 100 stars. Notably, nearly half of these repositories have over 1,000 stars, and 41.5% surpass the 10,000-star mark.

Finding 1: PBs in DS libraries have a significant impact on popular repositories, with 78% of affected repositories having over 100 stars, and 41.5 % exceeding 10,000 stars.

Implicit Impact. We identified and reported 44 potential performance issues in the dependent libraries, which have not been revealed. Their stars range from 14 to 37200, with a median of 362 and a first quartile of 152, showing their popularity. Over one-third of developers have responded and confirmed that the PBs affected their libraries. For example, in *LIT*, a developer acknowledged the PB in *to_dict* of *Pandas* affected the repository. He mentioned significant version updates are planned for *LIT*, and this issue will be included. This reveals there may be more significant yet unrecognized implicit impacts lie within the DS ecosystem. However, in some cases, simply updating the version is not a viable solution, especially when dependency constraints are present. For example, the developer of *tempo* pointed out that they cannot update *Pandas* across the entire project because of the need to align dependencies with *Databricks Runtimes*. They also mentioned lacking the time to devise a patch, especially since they no longer maintain the project.

Finding 2: Numerous performance issues in downstream libraries within the DS ecosystem remain unidentified, affecting many popular libraries. Some of them cannot be resolved by simply updating the versions of upstream libraries.

3.1.2 Symptoms Analysis

The taxonomy of symptoms is shown in Fig. 3. It includes three high-level categories: *Time*, *Memory*, and *Others*, along with 9 subcategories.

Time. This category covers PBs that exhibit abnormal time consumption, making up 74.8% (151/202) of all identified symptoms. Almost all time-related PBs can be attributed to *slow execution time*. We found that the execution time increased by an astonishing 158.83-fold, with extreme cases reaching up to 1,632-fold. Although such extreme cases are rare, their existence illustrates the severity of the PBs. Notably, the lower quartile showed a 4.75-fold increase in execution time, meaning 75% of cases experienced execution times more than five times longer. This is critical in core DS libraries, as their APIs are frequently used by dependent library developers. For instance, in *Pandas* issue *#44106* [18], users experienced a severe performance drop after upgrading to version *1.3.4*. Loading a large-column CSV file went from taking 1 minute to over 2 hours. This affected

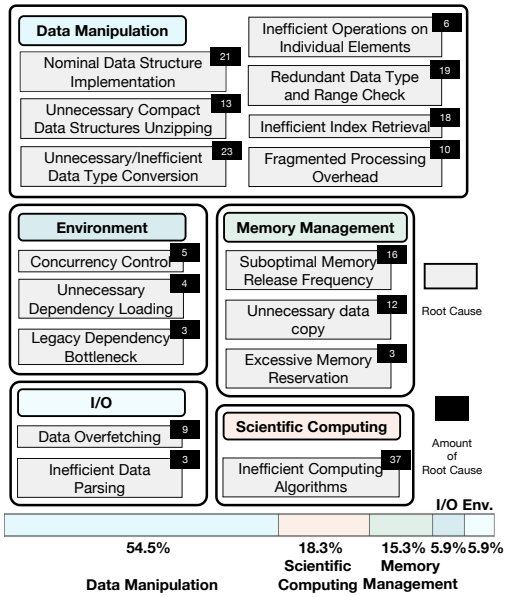


Fig. 4. Root Causes of PBs in DS Libraries. The five categories' percentages are shown on the bar below.

```
def append(self, other):
        """Append a collection of Index options together.
        Parameters
        other : Index or list/tuple of indices
        Returns
        Index
            The combined index.
        Examples
        >>> mi = pd.MultiIndex.from_arrays([['a'], ['b']])
        >>> mi
        MultiIndex([('a', 'b')],)
        >>> mi.append(mi)
        MultiIndex([('a', 'b'), ('a', 'b')],)"""
...
- label = self._get_level_values(i)
- appended = [o._get_level_values(i) for o in other]
- arrays.append(label.append(appended))
+ level_codes = [
+     recode_for_categories(
+         mi.codes[i], mi.levels[i], level_values, copy=False
+     )
+     for mi in ([self, *other])
+ ]
```

Fig. 5. Unnecessary Compact Data Structures Unzipping Example

```
def reset_identity(values)
...
    if ax.has_duplicates and not result.axes[self.axis].equals(ax):
-       result.index.get_indexer_non_unique(ax._values)
-       indexer = algorithms.uniqueId(indexer)
+       target = algorithms.uniqueId(ax._values)
+       indexer, _ = result.index.get_indexer_non_unique(target)
        result = result.take(indexer, axis=self.axis)
    else:
        result = result.reindex(ax, axis=self.axis, copy=False)
```

Fig. 6. Inefficient Index Retrieval Example.

both downstream libraries like *abhishekkrthakur/autoxgb* and users directly working with CSV files. Another category is *program halt*. In this case, the program entered an infinite loop and stopped responding. For example, the program will halt when a user attempt to assign a new column to a dataframe with a non-unique index.

Memory. This category covers PBs with unexpected memory use, accounting for 22.3% (45/202) of all cases. *Abnormal Memory Usage*, which represents the largest portion within this category, is observed in 31 (15.1%) of the PBs. It refers to situations where the used memory being significantly higher than expected under normal operations. Additionally, *Memory Leak* and *Out-Of-Memory* incidents account for 11 (5.4%) and 8 cases (4.0%), respectively. *Memory leak* occurs when memory usage continuously increases, while *Out-Of-Memory* situations occur when the system or application runs out of available memory, leading to failure or shutdown.

Others. This category includes symptoms that do not fall under the time or memory categories, making up 3.0% (6/202) of the total. Within this group, two cases involved GUI lagging, which refers to delays in the graphical interface that affect user experience. One case resulted in excessive performance warnings during execution, and another led to abnormal CPU utilization. Furthermore, three cases' symptoms are unknown.

Finding 3: Over 70% of the PBs decrease execution speed, with over 75% increasing running time by at least five times. Furthermore, over 20% PBs cause memory problems, such as *Memory Leak* and *Out-Of-Memory*.

3.2 Root Causes (RQ2)

The taxonomy of PB root causes is shown in Fig. 4. It is organized into five primary categories (i.e., *Data Manipulation, Memory Management, Scientific Computing, I/O* and *Environment*).

Data Manipulation. This category covers PBs encompassing inefficiencies and suboptimal practices regarding data manipulation, especially data structure. It accounts for 110 (54.5%) of the PBs.

Nominal Data Structure Implementation. Advancing technology and evolving user needs are driving DS libraries to integrate more efficient and varied data structures like tensors and extended arrays. Such introduction of new data structures requires adaptation to avoid PBs. Specifically, 21 (10.4%) of the PBs are caused by *Nominal Data Structure Implementation*. It refers to scenarios where libraries nominally support some data structure but lack specific and effective optimization methods. This shortfall often leads to a fallback on generic processing methods, which causes performance degradation due to the failure to unlock the data structure's full potential. This PB often arises in non-native structures, which are either provided by extension libraries or adapted via interfaces (e.g., *SciPy's sparse matrix* or *Pandas' ExtensionArray*), unlike native structures directly supported by DS libraries (e.g., *NumPy's ndarray*). Non-native data structures (e.g., *PyArrow's timestamp[ns][pyarrow]* in *Pandas*) often underperform compared to native ones (e.g., *Pandas datetime64[ns]*). For instance, in *Pandas* issue *#55031*, the groupby operation fell back to Python due to a *NotImplementedError*.

Unnecessary Compact Data Structures Unzipping. DS libraries provide various memory-efficient data structures, e.g. `sparse matrix` and `RangeIndex`. These data structures are prone to abnormal memory usage and long execution time when code unnecessarily expands them. This issue is categorized as *Unnecessary Compact Data Structures Unzipping* and it accounts for 13 (6.4%) of the PBs. For instance, in *Pandas* issue #53574 [22], Fig. 5 illustrates a PB that occurs when combining two `MultiIndex` objects. It is triggered by calling `_get_level_values(i)`, which unnecessarily extracted the full level values from the compact internal structure. `MultiIndex` is designed to store data efficiently using codes (integer mappings) and levels (unique values), but extracting full values unnecessarily expanded this compact structure, leading to significant performance degradation (a 20x slowdown). The original code densified the structure by appending these extracted values, as shown in the red section of Fig. 5. The optimized solution for this root cause is to preserve the compact data structure and prevent unnecessary unzipping. The optimized solution, depicted in green, avoids this by directly manipulating the codes and levels using `recode_for_categories`, preserving the compact data structure and preventing the unnecessary unzipping of the structure. Similarly, the optimized solution for this root cause follows the same approach—maintaining the compact structure and avoiding unnecessary expansion.

Inefficient Index Retrieval. The complexity of data and varying efficiency of index types across different data structures often cause great difficulties in selecting efficient indices. It causes *Inefficient Index Retrieval*, accounting for 18 (8.9%) of the PBs. The root cause refers to the suboptimal process of manipulating and accessing data indices, resulting in slower retrieval. It often occurs in index-heavy structures like *Pandas DataFrame* or *Series*, often due to redundant retrieval operations. An illustrative example is shown in Fig. 6, where the original code in *Pandas* inefficiently processed indices with duplicates by obtaining a non-unique indexer and then using `algorithms.unique1d` for each index value, including duplicates. This led to inefficient index retrieval. To enhance retrieval speed, the code was optimized by applying `algorithms.unique1d` to index values to ensure uniqueness, followed by indexer retrieval. This reduces repetitive tasks and ensures each index value is processed only once.

Inefficient Operations on Individual Elements. In DS libraries, which are typically optimized for handling large arrays and complex data structures, attention to individual elements involving scalar or single-element arrays is often lacking. These libraries use operations that work well for large or multi-dimensional data while inefficient for individual elements. We categorize this type as *Inefficient Operations on Individual Elements*, which accounts for 6 (3.0%) of all PBs. It is often found

in multi-element-focused structures (e.g., *NumPy arrays*, *Pandas Series*) when individual elements are accessed inefficiently. Although these inefficiencies may appear insignificant because they occur on individual elements, their impact can be substantial due to their frequent use in programs. In many cases, such operations are performed repeatedly within critical loops or heavily used functions, leading to cumulative delays that can significantly degrade overall performance. For example, issue *#19010* [8] in the *NumPy* project highlighted a significant performance inefficiency where calls to `np.empty(3)` were substantially slower than `np.empty((3,))`. The issue stemmed from the excessive overhead when managing individual elements, which are scalar values here. This inefficiency, particularly when generating thousands of empty arrays, can significantly slow down performance and be hard to detect during profiling.

While the above-mentioned categories mainly relate to *data structures*, our dataset reveals another set of PBs that are less about the structure itself but more about the *data types* of the individual elements within these structures.

Unnecessary/Inefficient Data Type Conversion. Data type conversion is a frequent operation in DS libraries. Converting integers to floats for numerical computations is a typical example. However, we fount not every data type conversion process is designed efficiently or even necessary, particularly with date conversions and boolean mask transformations. This issue, identified as *Unnecessary/Inefficient Data Type Conversion*, accounts for 23 (11.4%) of the PBs observed. For example, the *Matplotlib* issue *#23968* [12] demonstrates a rendering lag in larger windows, originating from an unnecessary data format conversion in the `paintEvent` function of `FigureCanvasQTAgg`. The conversion of image data from `rgba8888` to `argb32` using `np.take` led to increased processing times, especially with larger windows. This inefficiency was unnecessary, as `Qt` natively supports handling `rgba8888` format. By recognizing that `Qt` can directly process `rgba8888`, we can avoid the conversion and thus eliminate the associated delays.

Redundant Data Type and Range Check stands as another notable PB in DS libraries, accounting for 19 (9.4%) of all PBs. This issue often arises in iterative processes, leading to PBs. For example, in the *Scikit-learn* issue *#21242* [9], a performance degradation in `CountVectorizer.transform()` was traced back to a redundant type and range check. The check for uppercase characters in the vocabulary, initially conducted in every `transform` call, resulted in unnecessary repetition and a substantial slowdown. By moving this check to the `fit` method and only executing it during the initialization phase, the fix effectively eliminated the redundant checks. *Condition optimizing* (Sec. 3.4) is frequently used to solve this category of PB.

Fragmented Processing Overhead arises when multiple related data processing operations are executed separately, leading to inefficiencies that can be significantly reduced by combining these operations. It accounts for 10 (5.0%) of all PBs. For instance, building a `DataFrame` column by column in a loop (using `df[x] = v`) fragments the `DataFrame`, leading to inefficiency. Developers improved this by first compiling all data into a dictionary and then creating the `DataFrame` in one step, which significantly accelerates the process by eliminating column-by-column operations.

Scientific Computing. This category covers PBs related to scientific computing, accounting for 37 (18.3%) of PBs. It focuses on the performance of complex mathematical and numerical algorithms, such as optimization and interpolation. Inefficiencies here stem from how efficiently the algorithms execute calculations, not from how data is organized or manipulated in memory.

Inefficient Computing Algorithms involves developing algorithms that are poorly optimized for performance, particularly with complex or large data. This inefficiency typically arises from choosing suboptimal computational methods. An example of this is issue *#15150* [5] in *Scikit-learn* `RBFInterpolator`, where handling high-dimensional data was significantly slowed down due to inefficient computational methods. By replacing Python loops with optimized matrix operations like those in *BLAS*, computational efficiency was greatly enhanced.

Memory Management. While data manipulation issues typically slow down computations, memory management issues are equally crucial in terms of efficiency. These PBs can lead to inefficient resource use, causing increased memory consumption, memory leaks, or even out-of-memory errors. This category accounts for 31 (15.3%) of PBs. We further categorize these memory management issues into two subcategories.

Suboptimal Memory Release Frequency refers to a scenario where memory is not released efficiently, primarily due to reference cycles and unreleased references. It accounts for 16 (7.9%) of all PBs. This issue occurs when the garbage collector or memory management mechanisms are unable to reclaim memory that is no longer needed, often because the program still holds references to it. *Correct Reference Counting* (Sec. 3.4) is often used to solve this category of PB.

Unnecessary Data Copy is another PB which accounts for 12 (6.0%) of all PBs. For example, the use of `np.var(X, ddof=1, axis=0)` in *PCA transformations* creates temporary arrays to store intermediate differences and squared values, leading to extra memory consumption. This increases memory usage and slows execution due to redundant memory allocation and copying, which can significantly affect computation in memory-constrained environments or large-scale data processing tasks. In severe cases, it may cause memory exhaustion, resulting in system crashes.

Excessive Memory Reservation occurs when a program applies an aggressive memory preallocation strategy that reserves significantly more memory than needed. This leads to inefficient memory usage and, in extreme cases, out-of-memory (OOM) errors. It accounts for 3 (1.5%) of all PBs. For example, *NumPy* issue #18141 revealed how an update in v1.19.5 increased memory consumption due to changes in OpenBLAS's preallocation strategy. This led to excessive memory reservation per thread, causing OOM errors, particularly in virtualized environments like Docker.

I/O. This category is prevalent in scenarios involving data reading and writing operations, where inefficiencies can significantly impede the overall performance of DS workflows. *I/O* is limited to the data loading phase and do not involve PBs caused by data processing after data has been loaded. It accounts for 12 (5.9%) of PBs.

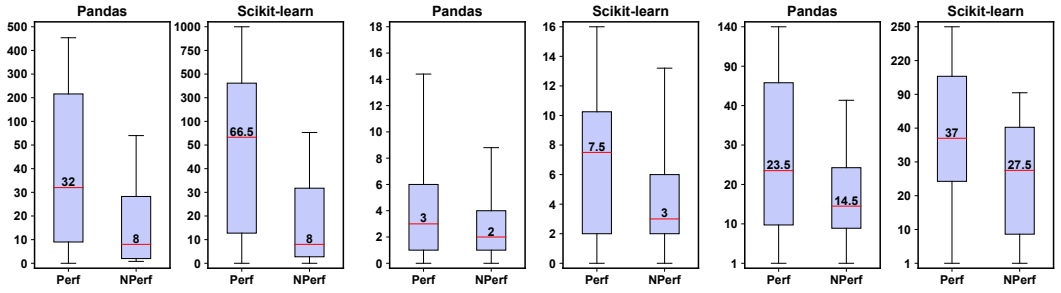
Data Overfetching occurs when a system or function retrieves more data than needed. It accounts for 9 (4.5%) of PBs. This typically happens when data loading mechanisms are not optimized to selectively load only the required data, resulting in increased resource consumption and delays. For example, in *Pandas* issue *#32727* [17], the `read_excel` function initially loads an entire Excel file into memory, even when a row limit is specified through the `nrows` parameter. This implementation is inefficient for large files, causing unnecessary memory usage and time delays, especially when the intention is to read a small fraction of the file.

Inefficient Data Parsing occurs when the reading and extracting data operation is not optimized, leading to slow I/O operations and excessive resource use. For example, in *Pandas* issue *#49929* [20], `read_html` struggled with poor performance due to an inefficient *XPath* expression. It causes extensive *DOM* traversals when parsing an HTML file with multiple tables. The fix streamlined the *XPath* to reduce traversals, improving the parsing speed by nearly 40 times.

Environment. This category, accounting for 12 (5.9%) of all PBs, arises from suboptimal utilization or configuration of external dependencies and system resources.

Concurrency Control refers to the inefficient management of the simultaneous execution of multiple threads, which leads to increased execution times and potential deadlocks due to improper synchronization and resource contention. It accounts for 5 (2.5%) of PBs. In *NumPy* issue *#24252* [13], a PB was caused by the inefficient handling of the *Global Interpreter Lock (GIL)* during multi-threaded concatenate operations. The solution involved releasing the *GIL* only for large arrays to minimize threading overhead.

Unnecessary Dependency Loading involves loading external libraries or dependencies not needed for specific functions. It accounts for 4 (2.0%) of PBs. In *Scikit-learn* issue *#26098* [15], excessive



(a) Bug open duration (days) (b) Number of bug comments (c) Patch size (lines of code changes)

Fig. 7. Comparison of root causes locating and bug-fixing effort

memory usage in estimators like `KNeighborsClassifier` was traced back to importing *Pandas* functions unnecessarily. The resolution involved replacing these imports with simpler checks to reduce memory load.

Legacy Dependency Bottleneck refers to PBs caused by the reliance on outdated external components, such as compilers or protocols. Despite enhancements to these dependencies, the library still performs poorly until the code is updated to exploit the improvements. It accounts for 3 (1.5%) of PBs. In *Pandas* issue *#23209* [11], developers initially used `libc isnan` function to handle floating-point equality checks as a workaround for a bug in *Microsoft Visual C++ (MSVC)*. This solution led to a loss in performance because it cannot be inlined. Once the *MSVC* bug was resolved, developers optimized *Pandas* by replacing the `isnan` workaround with direct equality checks.

Finding 4: Over half of the PBs are caused by inefficient operations on *Data manipulation*, particularly in *data structures*. *Memory management*, *Scientific computing*, and *I/O* contribute to more than a third of these issues. The variety of root causes complicates locating efforts. The identified root causes, which are highly domain-specific and distinct from general performance issues found in earlier studies, require new locating and fixing strategies.

3.3 Effort and Challenges (RQ3)

We first present the results regarding the effort involved in resolving PBs. The effort is measured using three key metrics: Bug Open Duration, Number of Bug Comments, and Patch Size (Sec. 2.3.3). It can be seen that resolving a PB in core DS libraries typically spans over one month. This process often involves numerous discussions, reaching up to 77 bug comments. Furthermore, on average, the patches for bug fixes tend to be extensive, which involves changes across 31 lines of code, highlighting the substantial effort required. Fig. 7 offers a comparative boxplot analysis of PBs and non-performance issues. It can be seen that tackling PBs demands more effort compared to non-performance related bugs. For instance, in *Scikit-learn*, the median duration for resolving PBs is 66.5 workdays, whereas only 8 workdays for non-performance issues.

Table 3 presents our U-test results (p-values) and Cliff's Delta values. Since all p-values are below 0.05, we reject the null hypotheses (Sec. 2.3.3) with over 95% confidence. Most Cliff's Delta values (except for bug comments in Pandas) are approximately or above 0.33, indicating medium effect sizes [53]. This shows that the difference in root causes and bug-fixing efforts between PBs and non-PBs is both statistically significant and practically meaningful. As for bug comments in *Pandas*, the Cliff's Delta value is 0.1756, a small effect size indicating no significant difference in discussion amounts between PBs and non-PBs [53]. Specifically, for issues with fewer than two comments (the median for non-PBs), the Cliff's Delta value is 0.0849. This shows that for lower-effort issues, the difference is negligible. However, for issues with comment counts above the median, which

Table 2. Locate Root Causes and Fix Performance Issue effort

Metric	Minimum	Median	Maximum	Mean
Bug Open Duration (days)	1	40	2666	277.47
Number of bug comments	0	4	77	6.44
Patch size (LOC)	1	31	2667	85.66

Table 3. *p*-values of Mann-Whitney U-tests

Subject	Bug open duration		Number of bug comments		Patch size	
	P-value	*Cliff's D*	*P-value*	*Cliff's D*	*P-value*	*Cliff's D*
Pandas	7.62e-8	0.4187	0.0231	0.1756	0.0003	0.2778
Scikit-learn	0.0015	0.4179	0.0148	0.3192	0.0259	0.3311

require more effort to resolve, the Cliff's Delta rises to 0.3553, indicating a medium effect size. This suggests PBs have more comments than non-PBs for issues requiring more effort. Thus, while overall discussion amounts may seem similar, PBs demand more discussion for complex issues.

Finding 5: Compared to non-performance bugs within the same libraries, PBs have more comments, larger patches, and longer lifetimes. This underscores the need for developer expertise in locating PBs' root causes in DS libraries. Given the limited number of developers and constrained resources, PBs' root causes locating and fixing tools in DS libraries are needed.

We further looked into bug comments and bug-fixing patches to understand the difficulties in locating PBs' root causes and fixing them in DS libraries.

Root Causes Locating Challenges. We analyzed issues that were closed after one month with over 10 comments and identified 3 challenges in locating popular DS libraries' PBs' root causes.

Hardware Related. Developers often spend large time locating hardware-related PB root causes. For instance, in *Numpy* issue *#17989*, the native *ARM* code runs ten times slower than the translated *x86-64* code on *Apple M1* chips. After 20 discussions involving six developers, it was discovered that the native *ARM* code did not fully utilized the *NEON* instruction set. Initially, the developers did not fully understand the cause of the PB, which led to extensive discussions and tests.

Limited understanding of underlying data structures and their implementation. Locating PBs' root causes often becomes complicated due to the lack of deep understanding of the libraries' underlying data structures and their implementation. For example, in *Pandas* issue *#52070* [21], developers initially had a limited understanding of the conversion from `ArrowArray` to `MaskedArray`, as well as the data type mapping between *Pyarrow* and *Pandas*. It led to over ten rounds of discussion among three members before identifying the root cause and proposing a solution.

User-needs misalignment. Another challenge is the discrepancy between developers' understanding of user practices and actual user behaviors. Developers' delay in understanding user needs often extends locating time. For instance, in *Pandas #30552* [16], developers initially failed to predict the significant memory increase, triggered by `groupby` operations on `Categorical` data with numerous unused categories. While making `observed=True` the default setting offered a solution, there was hesitation due to concerns over compatibility with *Pandas* design principles. Ultimately, after more than 20 discussions, developers acknowledged the real user needs and fixed it.

Fixing Challenges. To understand why fixing efforts are substantial in popular DS libraries, we manually reviewed PBs with LOC changes exceeding the upper quartile and identified four challenges.

Refactoring from High-Level Abstractions to Low-Level Operations. Developers often encounter challenges when shifting from high-level data abstractions to lower-level ones, which causes extensive code refactoring. For example, in *Pandas* issue *#24562* [14], the code was refactored to shift from using the high-level `datetime64[ns]` abstraction to the more granular `int64` representation. This transition was intended to enhance performance by operating directly on the underlying integer representation of timestamps, which count nanoseconds since the UNIX epoch. However, this change resulted in a substantial increase in LOC and code complexity. Developers had to introduce additional steps to manage the conversion between `int64` and `datetime64[ns]`, which were previously abstracted away. The shift also necessitated manual handling of time calculations, boundary conditions, and edge cases that were previously managed by the higher-level abstraction.

Extending logic for new data structure support. In DS library, developers need to extend support for new data structures, which often leads to increased complexity in the processing logic. In *Pandas* issue *#44240* [19], to improve performance, developers expanded the factorization logic to support a wider range of data types, including `ExtensionArray` and `BaseMaskedArray`. They introduced a `_factorizers` dictionary to map data types to the appropriate factorizer class, enabling dynamic selection based on the data type. Additionally, they implemented specific logic to handle mask data during the factorization of `BaseMaskedArray` types.

Managing and Refactoring Block Operations for Memory Optimization. This challenge focuses on expanding and refactoring the management logic for block operations to optimize memory usage. In *Pandas*, a "block" is a fundamental data structure that holds chunks of data, crucial for efficient storage and manipulation. One specific example involves the optimization of the delete operation in data blocks to avoid unnecessary data copying. Originally, the delete method would remove data and return a new block, often requiring a full data copy. The optimized approach, however, avoids this by splitting the block into multiple smaller blocks, which share the original data without duplicating it. This change reduces memory overhead but requires additional logic to manage these multiple blocks correctly. To implement this, the developers introduced a `_factorizers` dictionary and made substantial adjustments to the `BlockManager` logic to handle multiple blocks returned by the delete operation. This included updating the management of block lists, position indices, and block numbers to ensure data structure consistency after deletions. The necessity to handle both one-dimensional and two-dimensional data further increased the complexity of the code, adding more conditions and specific handling paths.

Refactoring with Lower-Level Languages. This challenge involves refactoring existing code to use lower-level languages for performance improvements. For example, in issue *#15527* [6], *Scikit-learn* refactored the `_dump_svmlight` function from Python to Cython. Originally implemented in Python, the method was effective but faced performance issues with large datasets. To resolve this, developers introduced a Cython-based method, `_dump_svmlight_file`, which optimized performance by reducing Python overhead and improving data handling speed. However, this also increased code complexity.

Finding 6: The challenges of root causes locating and fixing PBs in DS libraries arise mainly from complex data structures and the extensive refactoring they require.

3.4 Common Fixing Strategy (RQ4)

Also, we found there are about 20% of fixes has LOC not larger than 10, which indicates they can be fixed through simple changes. We then manually checked the patch and found several fixing strategies with small LOC that can be automated.

Conditions Optimizing. This fix strategy improves performance by refining condition checks to avoid unnecessary operations. For instance, in *Pandas*, the code checks if the old and new values are

identical and if the operation is in-place. If these conditions are met, it skips redundant actions like memory copying or block-splitting, thereby returning early and improving execution efficiency.

Correct Reference Counting. This strategy ensures proper memory management in Python by accurately adjusting reference counts for objects. In Python, reference counting tracks how many references point to an object, and when this count drops to zero, the object is deallocated. Errors in reference counting can lead to memory leaks or crashes. This strategy is common in Python's C extensions and is relatively easy to fix by ensuring that `Py_INCREF` and `Py_DECREF` calls are correctly paired.

Cache by Adding Decorators. The strategy improves performance by using decorators to store results of expensive computations, which avoids the need to recalculate them on subsequent calls. Decorators like `@cache_readonly` wrap functions or properties with a caching mechanism, allowing the return of cached values instead of recalculating them each time. For instance, in *Pandas*, the `@cache_readonly` decorator is applied to the `data_index` property, caching its value after the first calculation and reducing the overhead on future accesses.

Parameter Modification. The strategy involves altering method parameters to enhance performance. It is straightforward to implement, as it typically involves changing or adding a parameter value. For example, in `MultiIndex.copy`, a method used to duplicate index structures in *Pandas*, a new parameter called `keep_id` was introduced to control whether the new index retains the `_id` of the original index. The `_id` serves as a unique identifier for the index. When making a shallow copy (i.e., `deep=False`), setting `keep_id=True` allows the new index to share the original index's `_id`, avoiding the need to recalculate this identifier. This reduces redundant work during shallow copies, thereby improving performance.

Threshold Adjustment. The strategy optimizes performance by adjusting thresholds or limits within the code to reduce unnecessary processing or better handle high workloads. For example, before the change, references were cleared on every call, leading to slowdowns. The updated approach introduces a dynamic `clear_counter` adjusting based on the number of active references. If references fall below half the threshold, the counter decreases; if they exceed the threshold, it increases.

4 Implications, Application and Threats

In this section, we first highlight the implication of our study for developers and researchers, then showcase the application in PB detection and conclude with a discussion on potential risks.

4.1 Implications

Our study shows that PBs seriously affect the downstream repositories in DS ecosystem (Finding 1,2). In this section, we discuss implications for new research to combat PBs in popular DS libraries. To effectively handle PBs in DS libraries, existing tools need significant enhancements in both detection and fixing capabilities.

4.1.1 Detecting PBs' Existence in DS Libraries.

First, effective techniques for detecting PBs are crucial in DS libraries. Our study reveals common symptoms (Finding 3) of PBs that developers should monitor when testing and running DS libraries to detect potential issues. Half of these symptoms (i.e., *Slow Execution Time*, *Out of Memory*, *Memory Leak*, and *Abnormal Memory Usage*) can serve as credible oracles for identifying PBs in DS libraries. Furthermore, our study identifies common root causes (Finding 4) of PBs that can aid developers in diagnosing these issues. Some of these root causes exhibit unique characteristics, particularly when compared to those found in other domains [32, 35, 49, 57, 70] or general studies [39, 55]. These distinct root causes can be leveraged to test PBs by generating testing oracles. For instance, in the

category of *Data Manipulation*, subcategories like *Unnecessary Compact Data Structures Unzipping* demonstrate strong DS-specific features. When investigating performance degradation caused by this root cause, such as converting a *RangeIndex* to an *Int64Index*, developers can monitor the index type before and after an operation. If a compact data structure changes inappropriately (e.g., from *RangeIndex* to *Int64Index* when not needed), this is flagged as a potential performance bottleneck.

To automate this process, developers can integrate these checks into existing testing frameworks like *pytest* or *unittest*. For example, a custom *pytest* fixture could monitor the type of indices in *DataFrame* objects, and raise an alert if a type change occurs during an operation, flagging it as a potential performance issue.

4.1.2 Locating PBs' Root Causes in DS Libraries.

Second, tools for locating PBs in DS library are also needed to ease the developers' debugging effort. We identified two major challenges to locate the root cause: the lack of certain specialized knowledge, such as hardware-related aspects and a limited understanding of underlying data structures and their implementations. This knowledge is often library-specific and difficult to acquire. Therefore, researchers should focus on developing tools that assist developers in locating root causes.

There are two stages in the process of root cause location. Firstly, the relevant knowledge should be extracted and provided. To acquire specialized knowledge, relevant technical documents should be searched, including both hardware-related aspects and underlying data structures. Some comments from previous issues or certain past PRs are also crucial for root cause location, as we found developers sometimes refer to previous discussions where the PB were talked about but overlooked, and some PBs are caused by code modifications in earlier PRs. Retrieval-Augmented Generation (RAG) can be used here to retrieve relevant documents and generates context-aware knowledge [38, 72]. This allows it to handle complex, domain-specific issues such as finding hardware configurations and data structures related knowledge. Additionally, RAG can stay current with the latest knowledge. This makes it more adaptable and precise in evolving DS libraries.

After acquiring relevant knowledge, developers must reason through potential root causes by manually studying related knowledge like data structure implementations, and analyzing corresponding code through relating it to the knowledge. This process also requires determining whether issues are general or limited to edge cases. The effort needed to understand code and correlate it with prior knowledge is substantial, making PB diagnosis particularly challenging for contributors unfamiliar with the repository. Previous LLMs struggle with short chains of thought and miss clear, traceable reasoning paths. Also, they lack the ability to break down problems. As a result, when analyzing PBs where multi-step reasoning is required, these models often failed to reach correct conclusions. Furthermore, the absence of explicit reasoning paths makes it difficult for developers to pinpoint errors or find valuable clues for further manual analysis. Recent advances in reasoning models and frameworks (e.g., GPT-O1 [23], DeepSeek-R1 [48], and STaR [84]) can enable the automation of this process by introducing address these limitations by introducing long-chain, multi-step reasoning architectures with explicit reasoning processes.

4.1.3 Fixing PBs in DS Libraries Automatically.

Researchers should develop tools to automate PBs with simple fixes. We found 20% of PBs require small effort to fix, with five common strategies identified, all involving minor changes with straightforward logic. Furthermore, for PBs requiring significant fixing efforts, researchers could explore the use of LLMs for aiding fixes. Three of our identified challenges are related to refactoring, primarily driven by the need to adopt more efficient elements, such as data structures and programming languages. These refactorings often involve operations on low-level data structures,

which are particularly challenging in the context of DS library performance enhancements. This difficulty arises because modifying these underlying structures typically requires extensive changes throughout the codebase, as well as deep expertise in both the original and target data structures to ensure the refactorings improve performance without introducing new issues. For example, modifying data structures often requires not only adapting existing functions to work with the new structures but also ensuring that these changes propagate correctly throughout the system. This process can be extremely labor-intensive and error-prone, particularly in DS libraries where performance optimization is deeply intertwined with the underlying data structures. Additionally, changing the programming language of certain modules adds another layer of complexity, as it may necessitate a complete rethinking of how key components interact to maintain compatibility and efficiency in the new environment. One potential research direction is developing LLM-based tools that can analyze the existing codebase, identify inefficient data structures, and suggest the code to use more efficient alternatives. For practitioners, prioritising fixes for PBs with wide downstream or critical impacts is essential. Furthermore, we found version compatibility issues often hinder downstream developers from upgrading to patched versions. In such cases, practitioners should consider backporting PB fixes to widely-used older versions to minimise disruption.

4.2 Application

To demonstrate the usefulness of our findings, we derived one metamorphic relation and one testing oracle from our curated root causes (*Nominal Data Structure Implementation* and *Unnecessary Compact Data Structures Unzipping*), to test core DS libraries.

Array Interface Performance Equivalence. The metamorphic relation to identify potential PBs is designed by evaluating the execution time of identical operations on data structures with an array interface (e.g., *tensors* and *JAX*) against *NumPy* arrays. It is derived from *Nominal Data Structure Implementation*. This relation operates on the premise that, given the similarity in the operational interfaces provided by these data structures and the extensive performance optimizations already applied to *NumPy*, operations executed on these data structures should exhibit similar execution time.

Compact Storage Efficiency. Specifically, data indexed in a compressible format should avoid instantiation into more memory-intensive structures. For instance, a `RangeIndex` for an arithmetic sequence with a step of 1 can be efficiently represented by just three numbers. This oracle guides the avoidance of unnecessary memory consumption by leveraging data structure compression capabilities.

Results. We conducted tests on ten cases within DS libraries (e.g. *Pandas*, *Scipy*, and *Matplotlib*), and reported the findings to the developers. Four cases were confirmed as PBs needing fixes while four other issues were identified as user adaptation issues, such as value-dependent behaviors. The last two were found to be anomalies specific to our test environment.

4.3 Threats

The validity of our study faces *external threats* primarily due to the data used. Our study focuses on seven DS libraries written mainly in Python. Consequently, it is uncertain if our results are applicable to DS libraries in other languages, such as R, warranting further investigation into language-based differences. Furthermore, our study analyzes PBs from Github issues while StackOverflow is another valuable source of PBs. It is interesting to further explore PBs from StackOverflow to strengthen our findings. However, considering that we have already spent four person-months analyzing 202 performance bugs, further exploring PBs from StackOverflow would require significant manual effort. The main *internal validity* threats stem from our manual study process. To address potential inaccuracies and biases, two authors (each with over three years of experience) independently

categorized the data using the general open-coding scheme, mirroring practices from similar studies. Any discrepancies were resolved through discussions with a third author, ensuring consistent labeling across the study.

5 Related Works

5.1 Empirical Study in Data Science (DS)

DS has become an emerging field in recent years. Several studies have aimed to explore and understand the DS procedure. Roh et al. [68] examined data collection techniques in big data, presenting a workflow that explains how to improve data or models in a machine learning system. Another study identified the software engineering practices and challenges in developing AI applications within Microsoft teams, highlighting key differences between AI software processes and those in other domains. They proposed a nine-stage workflow for DS software development [30].

A few other studies have targeted DS libraries. Tao et al. [76] studied how performance concerns are documented in DS libraries, finding that official documentation and crowd-sourced documentation (e.g., GitHub) complement each other on performance-related content. They also observed that maintenance of performance-related documentation is relatively stagnant and peripheral compared to the active evolution of the underlying APIs. Ahmed et al. [29] examined the bug characteristics of data analytics programs in R and Python, categorizing bug types, root causes, and their impact on program execution. Islam et al. [54] investigated the bug characteristics and fix patterns in popular Data Science libraries like Caffe, Keras, Theano, and Torch. Tao et al. [75] also empirically analyzed the causes and solutions for "bad" error messages by qualitatively studying 201 Stack Overflow threads and 335 GitHub issues. Despite many studies mentioning that performance is critical in DS, little attention has been received to understanding PBs in popular DS libraries.

5.2 Performance Bugs (PBs)

Studies show that PBs occur frequently [55] and consume a significant portion of developer time [83]. Many empirical studies have characterized PBs from various perspectives, such as root causes, detection, diagnosis, fixing, and reporting, across different domains. These studies include desktop and server applications [55, 62, 67, 74, 83], highly configurable systems [49, 51], mobile applications [56, 57], database-backed web applications [81, 82], JavaScript systems [70], and Deep Learning systems [35]. Jin et al. [55] studied the root causes of PBs in several PC and server-side applications and identified efficiency rules for detecting these bugs. Similarly, Pei et al.[67] analyzes how PBs are discovered, reported, and fixed, comparing them with non-PBs in popular codebases like Eclipse JDT, Eclipse SWT, and Mozilla. Song et al.[74] extends this analysis by exploring how performance problems are observed and reported by real-world users, and examining the applicability of statistical debugging for performance issue diagnosis. Liu et al. explored the manifestation, locating, and fixing effort of PBs in smartphone applications, identifying common root causes. More recently, Cao et al. [35] investigated performance problems in Deep Learning systems developed in TensorFlow and Keras, characterizing their symptoms, root causes, and stages of introduction and exposure.

While previous studies in these domains have examined the *Root Causes*, *Locating&Fixing challenges* associated with PBs, the characteristics observed in DS libraries present different challenges that have not been fully explored. Furthermore, they have not performed an impact scope analysis due to the characteristics of their study objects, which tend to focus more on applications or frameworks at the end of the PB propagation chain. Our study fills the gap.

Root Causes. The root causes of PBs in DS libraries differ significantly from those in non-DS PB studies [35, 51, 55–57]. Over half of DS library PBs stem from inefficiencies in data processing operations, particularly issues tied to data structures, such as suboptimal indexing. These root causes

are largely absent in non-DS studies. Even when high-level categories like memory inefficiency or I/O bottlenecks appear similar, as noted in works like [70] or [31], the underlying subcategories' details and granularity differ significantly. It is noted that high-level classifications provide limited actionable insights for automated debugging. For example, while [31] include *Memory Inefficiency* categories like *memory leak* or *miscellaneous inefficiencies*, these classifications are too general to guide developers or researchers effectively. In contrast, our finer-grained categories, such as *suboptimal memory release frequency*, offer actionable insights specific to PBs in DS libraries. Similarly, while *inefficient algorithms/data structures* or *inefficient I/O* are noted in other studies, they often lack detailed explanations of underlying issues. Our classifications explicitly detail DS-specific characteristics, such as *Unnecessary Compact Data Structures Unzipping*, which highlights how unnecessary code expansions lead to abnormal memory usage and long execution time, with examples like *rangeindex* and *sparse matrix*. These detailed classifications make our taxonomy more actionable and domain-specific compared to non-DS PB studies.

Locating & Fixing Challenges. Challenges in locating and fixing PBs in DS libraries are rarely discussed in existing studies. For instance, among the related works [35, 49, 51, 55–57, 62, 67, 70, 74, 81–83], only [57] mentions debugging challenges caused by multiple threads or processes. DS libraries face unique challenges (e.g. *Refactoring from high-level abstractions to low-level operations*), as highlighted in Sec. 3.3. These challenges arise from the inherent complexity of DS libraries, such as handling intricate data structures and indexing structure.

Impact Scope. PBs in DS libraries differ significantly from others in impact scope. Prior studies [55, 62, 67, 74, 83] on PBs primarily focus on single projects, which lack extensive downstream dependencies, such as desktop applications. In contrast, our analysis indicates that PBs in DS libraries propagate widely through dependency chains and affect many popular downstream projects. In RQ1, our study confirms the severity of this propagation. We found that 78% of repositories dependent on DS libraries have over 100 stars, indicating their importance in the community. This highlights the potential for DS library PBs to disrupt popular downstream projects.

6 Conclusion

This paper presents the first comprehensive study to characterize PBs in seven popular DS libraries. We found that PBs in these DS libraries have a wide and severe impact on the DS ecosystem. Our research presents a detailed taxonomy of PB symptoms and root causes within these popular core libraries. We discovered that resolving PBs demands greater effort and expertise than addressing non-performance issues. We also found that the challenges in locating PBs' root causes and fixing them in DS libraries mainly arise from the complexity of specific data structures and the significant refactoring needed when making modifications. Furthermore, around 20% of the PBs could be fixed using simple strategies, suggesting the potential for automated repair approaches. Our findings offer practical insights for both developers and researchers on preventing and identifying PBs in DS libraries. In addition, we developed two oracles from our curated root causes and detected new PBs in DS libraries.

7 Data Availability

The dataset is archived in https://doi.org/10.5281/zenodo.15250091.

8 Acknowledgement

This paper was supported by the Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010145) and the Shenzhen Science and Technology Program (Shenzhen Key Laboratory Grant No. ZDSYS20230626091302006).

References

[1] [n. d.]. 8 Best Python Libraries For Data Science. https://www.bairesdev.com/blog/best-python-libraries-data-science/. Accessed: [2024.02.17].
[2] [n. d.]. Freqtrade. https://github.com/freqtrade/freqtrade. Accessed: [2023.12.17].
[3] [n. d.]. Freqtrade causing my VPS to crash. https://github.com/freqtrade/freqtrade/issues/3016. Accessed: [2023.12.17].
[4] [n. d.]. https://github.com/mwaskom/seaborn. https://github.com/scikit-learn/scikit-learn/issues/15527. Accessed: [2023.12.17].
[5] [n. d.]. Issue 15150 in Scikit-learn. https://github.com/scikit-learn/scikit-learn/issues/15150. Accessed: [2023.12.17].
[6] [n. d.]. Issue 15527 in Scikit. https://github.com/scikit-learn/scikit-learn/issues/15527. Accessed: [2023.12.17].
[7] [n. d.]. Issue 18921 in Scipy. https://github.com/scipy/scipy/issues/18921. Accessed: [2023.12.17].
[8] [n. d.]. Issue 19010 in Numpy. https://github.com/numpy/numpy/issues/19010. Accessed: [2023.12.17].
[9] [n. d.]. Issue 21242 in Scikit. https://github.com/scikit-learn/scikit-learn/issues/21242. Accessed: [2023.12.17].
[10] [n. d.]. Issue 22949 in Numpy. https://github.com/numpy/numpy/issues/22949. Accessed: [2023.12.17].
[11] [n. d.]. Issue 23209 in Pandas. https://github.com/pandas-dev/pandas/issues/23209. Accessed: [2023.12.17].
[12] [n. d.]. Issue 23968 in Matplotlib. https://github.com/matplotlib/matplotlib/issues/23968. Accessed: [2023.12.17].
[13] [n. d.]. Issue 24252 in Numpy. https://github.com/numpy/numpy/issues/24252. Accessed: [2023.12.17].
[14] [n. d.]. Issue 24562 in Pandas. https://github.com/pandas-dev/pandas/issues/24562. Accessed: [2023.12.17].
[15] [n. d.]. Issue 26098 in Scikit. https://github.com/scikit-learn/scikit-learn/issues/26098. Accessed: [2023.12.17].
[16] [n. d.]. Issue 30552 in Pandas. https://github.com/pandas-dev/pandas/issues/30552. Accessed: [2023.12.17].
[17] [n. d.]. Issue 32727 in Pandas. https://github.com/pandas-dev/pandas/issues/52016. Accessed: [2023.12.17].
[18] [n. d.]. Issue 44106 in Pandas. https://github.com/pandas-dev/pandas/issues/44106. Accessed: [2023.12.17].
[19] [n. d.]. Issue 44240 in Pandas. https://github.com/pandas-dev/pandas/issues/44240. Accessed: [2023.12.17].
[20] [n. d.]. Issue 49929 in Pandas. https://github.com/pandas-dev/pandas/issues/49929. Accessed: [2023.12.17].
[21] [n. d.]. Issue 52070 in Pandas. https://github.com/pandas-dev/pandas/issues/52070. Accessed: [2023.12.17].
[22] [n. d.]. Issue 53574 in Pandas. https://github.com/pandas-dev/pandas/issues/53574. Accessed: [2023.12.17].
[23] [n. d.]. OpenAI O1. https://openai.com/o1/. Accessed: [2025.02.20].
[24] [n. d.]. Pokémon Go makers call for calm as servers crash across Europe and US. https://www.theguardian.com/technology/2016/jul/16/pokemon-go-server-crash-niantic-europe-us. Accessed: [2023.12.17].
[25] [n. d.]. Python Libraries for Data Science: A Comprehensive Guide. https://www.codemotion.com/magazine/languages/python-libraries-data-science/. Accessed: [2024.02.17].
[26] [n. d.]. Requests. https://github.com/psf/requests. Accessed: [2023.12.17].
[27] [n. d.]. Taylor Swift's 'Midnights' album crashes Spotify, leaving fans shocked; nearly 8,000 outages reported. https://www.foxbusiness.com/entertainment/taylor-swifts-midnights-album-crashes-spotify-fans-shocked-8000-outages-reported. Accessed: [2023.12.17].
[28] [n. d.]. Top 10 Python Libraries for Data Science in 2024. https://www.geeksforgeeks.org/python-libraries-for-data-science/. Accessed: [2024.02.17].
[29] Shibbir Ahmed, Mohammad Wardat, Hamid Bagheri, Breno Dantas Cruz, and Hridesh Rajan. 2023. Characterizing Bugs in Python and R Data Analytics Programs. *arXiv preprint arXiv:2306.08632* (2023).
[30] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.
[31] Md Abul Kalam Azad, Manoj Alexender, Matthew Alexender, Syed Salauddin Mohammad Tariq, Foyzul Hassan, and Probir Roy. 2024. PerfCurator: Curating a large-scale dataset of performance bug-related commits from public repositories. *arXiv preprint arXiv:2406.11731* (2024).
[32] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. 2023. An Empirical Study of High Performance Computing (HPC) Performance Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 194–206.
[33] Ekaba Bisong and Ekaba Bisong. 2019. Matplotlib and seaborn. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners* (2019), 151–165.
[34] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines. (2022).
[35] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 357–369.
[36] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? Pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–12.

[37] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–31.
[38] Tianyu Chen, Lin Li, Liuchuan Zhu, Zongyang Li, Xueqing Liu, Guangtai Liang, Qianxiang Wang, and Tao Xie. 2023. VulLibGen: Generating names of vulnerability-affected packages via a large language model. *arXiv preprint arXiv:2308.04662* (2023).
[39] Yiqun Chen, Stefan Winter, and Neeraj Suri. 2019. Inferring performance bug patterns from developer commits. In *2019 IEEE 30th international symposium on software reliability engineering (ISSRE)*. IEEE, 70–81.
[40] Nadia Daoudi, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. A two-steps approach to improve the performance of android malware detectors. *arXiv preprint arXiv:2205.08265* (2022).
[41] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 2020. Characterizing the evolution of statically-detectable performance issues of android apps. *Empirical Software Engineering* 25 (2020), 2748–2808.
[42] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–34.
[43] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 509–519.
[44] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1435–1446.
[45] Paul D Ellis. 2010. *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge university press.
[46] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151.
[47] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, Chen, and Qi Alfred. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 385–396.
[48] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[49] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[50] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.
[51] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. Cp-detector: Using configuration-related performance properties to expose performance bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 623–634.
[52] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.
[53] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, Vol. 1. Citeseer.
[54] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.
[55] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.
[56] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in android apps. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 352–361.
[57] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.
[58] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[59] Florian Markusse, Alexander Serebrenik, and Philipp Leitner. 2023. Towards Continuous Performance Assessment of Java Applications With PerfBot. In *2023 IEEE/ACM 5th International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 6–8.
[60] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
[61] W McKinney. 2010. Data Structures for Statistical Computing in Python. 56–61. In *Proc 9th Python Sci Conf (SCIPY 2010)*.
[62] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 237–246.
[63] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 562–571.
[64] Fariha Nusrat, Foyzul Hassan, Hao Zhong, and Xiaoyin Wang. 2021. How developers optimize virtual reality applications: A study of optimization commits in open source unity projects. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 473–485.
[65] Jibesh Patra and Michael Pradel. 2022. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *Proceedings of the 44th International Conference on Software Engineering*. 1469–1481.
[66] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[67] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.
[68] Yuji Roh, Geon Heo, and Steven Euijong Whang. 2019. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering* 33, 4 (2019), 1328–1347.
[69] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.
[70] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.
[71] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.
[72] Shamane Siriwardhana, Rivindu Weerasekera, Elliott Wen, Tharindu Kaluarachchi, Rajib Rana, and Suranga Nanayakkara. 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics* 11 (2023), 1–17.
[73] Jigar Solanki, Laurent Réveillère, Yérom-David Bromberg, Bertrand Le Gal, and Tégawendé F Bissyandé. 2013. Improving the performance of message parsers for embedded systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1505–1510.
[74] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices* 49, 10 (2014), 561–578.
[75] Yida Tao, Zhihui Chen, Yepang Liu, Jifeng Xuan, Zhiwu Xu, and Shengchao Qin. 2021. Demystifying "bad" error messages in data science libraries. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 818–829.
[76] Yida Tao, Jiefang Jiang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. 2020. Understanding performance concerns in the api documentation of data science libraries. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 895–906.
[77] Yida Tao, Shan Tang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. 2019. How do api selections affect the runtime performance of data analytics tasks?. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 665–668.
[78] Saeid Tizpaz-Niari, Pavol Černỳ, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 189–199.
[79] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.
[80] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 20–31.
[81] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference*

on Software Engineering. 800–810.

[82] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. 2019. View-centric performance optimization for database-backed web applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 994–1004.

[83] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2012. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 199–208.

[84] Eric Zelikman, Yuhuai Wu, and Noah D Goodman. 2022. STaR: Self-Taught Reasoner. *arXiv preprint arXiv:2203.14465* (2022).

[85] Yutong Zhao, Lu Xiao, Andre B Bondi, Bihuan Chen, and Yang Liu. 2022. A Large-Scale Empirical Study of Real-Life Performance Issues in Open Source Projects. *IEEE Transactions on Software Engineering* 49, 2 (2022), 924–946.

[86] Yutong Zhao, Lu Xiao, Xiao Wang, Lei Sun, Bihuan Chen, Yang Liu, and Andre B Bondi. 2020. How are performance issues caused and resolved?-an empirical study from a design perspective. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 181–192.

Received 2025-02-26; accepted 2025-04-01