A Study of Bugs Found in the Moby Configuration Management System

Mattia Bonfanti
Supervisor(s): Prof. Dr. Ir. Diomidis Spinellis, Thodoris Sotiropoulos
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The study of bugs can provide important information to understand their nature in the context of complex software systems as well as supporting developers in their detection, fix and prevention. Previous studies focused on analyzing bugs under different perspectives such as changes at code level, frequency, semantics, symptoms, root causes and reproducibility through test cases. Although these studies offer valid methodologies that can be applied in different areas of bugs analysis, literature suggests that very little focus has been aimed towards configuration management systems.

This research has the goal to provide a characterization of bugs in the Moby configuration management system, an open framework used to create container systems. A random sample of 100 collected Moby bugs is manually inspected and categorized by their (1) symptoms, (2) root causes, (3) impact, (4) fixes, (5) system dependency and (6) triggers. Some representative takeaways suggest that: bugs symptoms are mostly linked to a specific root cause, which means that bugs occur in certain areas of the system; the modular nature of Moby drives up the criticality of the bugs as each component needs to work correctly; bugs are overall hard to reproduce as only 24.5% of the fixes included a test case; developing an automatic tool that provides a historical distribution of bugs can support maintainers in their work by enhancing bugs prevention.

The research provides an analysis and categorization of bugs in the Moby configuration management system. The work adds a new perspective to the literature on the topics of both bugs analysis and configuration management systems and can be used as a starting point for further studies.

1 Introduction

Understanding bugs is important in order to observe the nature of complex systems [8]. If the origin of bugs is comprehended, software engineers can be educated such that a bug can be prevented or detected. This can be accomplished by eventually building heuristic tools that will enhance the quality of software systems [15].

Previously conducted research on the topic gives a set of tools that can be used as a starting point for this project. The work from Zhao et al. in 2017 provides an empirical methodology to conduct a study about the characteristics of change types in bug fixing code. The applied method helps gain deeper insights into change types under three perspectives: across projects, across domains and across versions [38]. Another relevant research by Wang et al. in 2018 provides an additional layer to add to the methodology that analyzes multi-entity bug fixes in terms of their frequency, composition, and semantic meanings. After extracting commonalities from the identified dependencies between bugs, it will also be possible to draw recurring patterns [36]. The work by Chaliasos et al. from 2021 focused on the classification of typing-related compiler bugs in terms of their symptoms, root causes, fixes size and reproducibility [8]. The study provides an empirical methodology that can be built upon in order to gather, filter and categorize bugs for Moby. However, previous works by Hintsch et al. [20] and by Ali et al. [1] suggest that bugs analysis methodologies have not been applied as thoroughly to containers oriented software or to configuration management systems.

This study aims to analyze and characterize previously reported bugs in the Moby configuration management system. Moby is an open framework that provides a modular configuration of many standard components that can be assembled to create specialized container systems [28]. This research focuses on categorizing the symptoms, the triggers, the impact and the fixes of bugs in Moby by answering the following research questions: (1) What are the symptoms of these bugs?, (2) What are the root causes of these bugs?, (3) What is their impact?, (4) How do developers fix these bugs?, (5) Are these bugs system dependent?, (6) What triggers these bugs?

Applying and expanding bugs analysis methodologies in the context of a configuration management system that focuses on containers can provide an additional perspective on how to study and categorize software bugs. This study contributes to providing additional resources to the literature about configuration management systems as well developing a heuristic approach that can serve as a starting point for further research on the topic.

In section 2 the research terms and concepts are introduced to support the reader in the understanding of the work. Section 3 discusses the research questions that the paper aims to answer and the research practices that were followed. In section 4 the main contributions that this research makes to the topic are outlined and the techniques used for bugs analysis are contextualized to containers oriented systems. Section 5 discusses the results obtained from the analysis and classification of bugs in Moby and draws common patterns between them. Section 6 highlights the main insights and takeaways from the research as well as possible threats to its validity. Section 7 draws a critical comparison of the performed work to existing related research. In section 8 the responsible research practices that were followed are

illustrated as well as how the obtained results can be reproduced. Finally, an overview of the findings and the key points of the research is provided in the conclusion as well as a discussion of future improvements.

2 Concepts

The goal of this research is to study and characterize bugs in the Moby configuration management system. Before discussing the research steps in detail, it is important to explain the main concepts and terminology that the work builds upon. This will give the reader a better understanding of the applied methodologies and results.

This section aims to provide a brief background of the main terminology that is used in the research. The concepts of configuration management systems and bugs analysis are described in subsections 2.1 and 2.2 respectively. Subsection 2.3 illustrates the key characteristics of Moby and container systems in general.

2.1 Configuration Management Systems

Configuration management systems are a key part of the engineering process of establishing and maintaining performance and functionalities of a product consistently throughout its life-cycle. To achieve these goals, automation is a key factor as it introduces checks and redundancies that help reduce human errors in the configuration of a system. This is very important when it comes to the configuration management of large systems as errors can compromise performance and lead to inconsistencies that impact business operations and security [30].

As the complexity of a software system increases, so does the difficulty of preventing, identifying and fixing possible errors [22]. It is therefore important to keep a clear track of the bugs that occur in a setting like a configuration management system. This will allow these software products to be safe to use, scalable and stable [12].

2.2 Bugs Analysis

In a software system, a failure is identified as a component that is not behaving as expected. The flaw of the component that caused the failure is defined as a bug. It is important to note that a bug in the source code does not necessarily lead to a failure as that part of the code might not be executed [4].

A comprehensive study of bugs relies on three main factors: bugs details reported by users and developers, open access to source code and detailed documentation of the system being analyzed [25]. The process of bugs analysis starts from gathering the reported bugs in a database where they can then be retrieved and classified. This allows software engineers to understand where the bugs are coming from and the issue that led to this flaw. The goal is to reduce the identified types of bugs in future phases of a project [31].

2.3 Moby

Moby is an open framework created by Docker that allows the creation of specialized container systems [28]. A container is a lightweight virtualization of an operative system that isolates and defines one or more processes. Inside a container, the processes can be viewed as the owners of the entire system [26]. Moby provides a modular set of many standard components and a framework for assembling them into custom platforms; the most popular example is Docker [29].

A study by Ahmet et al. in 2020 highlighted the hardship of properly identifying vulnerabilities in complex settings involving huge numbers of containers, without a proper methodology [16]. As container based systems are increasing in popularity, with a projection of 90% of enterprises running them in production by 2026 [19], it is important to provide more tools and heuristics that can be applied to improve bugs detection and prevention. This will also offer additional support to researchers as they will need to maintain a clear view on the evolution of containers based technologies [33].

3 Methodology

The process of studying bugs in the Moby configuration management system started with defining the research questions that needed to be answered. This was followed by planning the steps needed to carry out the work.

This section aims to illustrate the research questions and sub-questions that were defined (subsection 3.1) and the step by step action plan that led to their answers (subsection 3.2) and formed the research methodology.

3.1 Research Questions

With the goal of studying known bugs and trying to find commonalities among them in the configuration management system Moby, the research questions as listed below were answered. Each research question has been broken down into sub-questions, which supported the research towards a complete solution for the topic [14].

1. What are the symptoms of these bugs?
   (a) What are the most common symptoms of the bugs?
   (b) What are the most frequent bugs?
2. What are the root causes of these bugs?
   (a) Can we identify common patterns between bugs?
   (b) What are the similarities between bugs?
3. What is their impact?
   (a) What are the consequences of the bugs on the system?
   (b) How critical is the bug on the system?
4. How do developers fix these bugs?
   (a) What is the most common fixing strategy?
   (b) What is the protocol in place to make a fix effective?
5. Are these bugs system dependent, i.e., do these bugs manifest regardless of the state of the underlying system?
6. What triggers these bugs?
   (a) What are the circumstances of the bugs?
   (b) How are the bugs reproduced?

3.2 Research Methodology

The first phase of studying bugs in Moby was the data gathering process, which allowed the creation of a data-set that could be used for further steps. From a starting amount of 21378 issues and 22079 pull requests, the goal was to filter out the ones related to actual bugs and their fixes. The result of this process produced a single source of information where each bug is linked to its fix and such information provided insight for the research.

The first attempt of fetching bugs from the issue tracker in the Moby GitHub repository was based on using the provided predefined labels used to mark issues and pull requests. Two labels are associated with bugs and their fixes, "kind/bug" and "kind/bugfix" respectively. However this first iteration of data fetching produced disappointing results as only 374 pull requests were associated with the label "kind/bugfix" and 29 with the label "kind/bug". On the other hand, only 1 issue was labelled as "kind/bugfix" and 1555 as "kind/bug".

The discrepancy in the amount of results linked to bugs related labels led to a different approach. Instead of relying only on predefined labels, the filtering of issues and pull requests was based on keywords applied to their title, description and labels. The identification of these keywords was performed by analyzing the research topic and the research questions in order to define the search concepts and constraints. Such concepts and constraints represent words that limit the scope of the research and help streamline the information gathering process [13]. All the keywords used are listed in section 4 under subsection 4.1.

The keywords based approach led to a much more satisfactory result of 2539 issues and 2539 pull requests. The fact that both searches resulted in the same number of results supports the assumption that every issue that was found represented a bug and was linked to a fix through a pull request. The gathered data was used as input for the next phase of the research, the bugs analysis.

The second phase of studying bugs in Moby was the analysis of the gathered bugs, which is the key step to answer the research questions. The structure of this phase builds upon what is proposed by Chaliasos et al. in their previous work from 2021 [8]. The bugs analysis was done in five iterations of 20 bugs each. After each iteration, the categorization of the bugs was subject to peer review so the subjectivity aspect of this approach could be limited. The outcomes of this process were also stored to a central data source.

The result from the analysis represents a categorization of the bugs that accounts for the elements outlined in the research questions: symptoms, root causes, impact, fixes, system dependency and triggers. By answering the research questions, a holistic view of bugs was built by analyzing if they were present in all configuration management systems or only in some. This allowed the creation of a pattern for the analyzed bugs. The final results of the research were then compared with those of existing work to draw critical feedback and arguments for improvement.

4 Experimental Setup

As discussed in section 3, the study of bugs in Moby consisted in two phases: bugs collection and bugs analysis. Both steps were performed following dedicated procedures that created the basis for this research to be reproduced and expanded upon.

This section aims to illustrate the experimental setup by describing in detail the choices made during the data collection process (subsection 4.1) and the reasoning behind the data analysis (subsection 4.2).

4.1 Collecting Bugs

The process of collecting bugs had the goal of creating a bugs database of about 2000 entries that could be used for the next step of this research as well as becoming a starting point for future work. Moby issues and pull requests from GitHub, the most popular code repository [5], provided the necessary insight about bugs and the respective fixes. At the time the research was conducted, the Moby issue tracker counted 21378 issues and 22079 pull requests.

The first step of filtering the bugs and their fixes out of the full amount of issues and pull requests involved the use of two labels predefined by the Moby developers: "kind/bug" and "kind/bugfix". Table 1 shows the results of this label based query. The outcome was disappointing due to the large discrepancy between the amount of bugs fixes from closed pull requests (328) and the amount of bugs from closed issues (1163).

Filter	# Open	# Closed	Total
is:pr label:kind/bugfix	46	328	374
is:pr label:kind/bug	0	29	29
is:issue label:kind/bugfix	1	0	1
is:issue label:kind/bug	392	1163	1555

Table 1: Predefined labels based filtering for pull requests (pr) and issues. Own work.

As the labels based filtering introduced the impossibility of correctly mapping bugs to their fixes, an alternative solution was needed to retrieve information. As the core of this research revolves around "software bugs", the following synonyms were defined: bug, glitch, error, flaw and failure. Using these keywords, a broader filtering was applied to issues and pull requests by looking into their labels, titles and descriptions. Tables 2 and 3 show the results of this keywords based filtering for issues and pull requests respectively.

Filter	# Open	# Closed	Total
is:issue bug in:label,body,title	392	1163	1555
is:issue glitch in:label,body,title	3	2	5
is:issue error in:label,body,title	1424	6274	7698
is:issue flaw in:label,body,title	4	7	11
is:issue failure in:label,body,title	267	730	997

Table 2: Keywords based filtering for issues. Own work.

From table 2 it can be seen that the good results obtained by the "kind/bug" label applied to the issues are still present. On top of that, the use of synonyms allowed the retrieval of additional bugs which were described by them. However, the keyword "error" provided a much higher amount of results as it is quite generic and the results associated with it also overlapped with ones from other keywords. This was taken into account during the issues filtering process and the "error" keyword was given lower priority than the other ones.

Filter	# Open	# Closed	Total
is:pr bug in:label,body,title	68	509	577
is:pr glitch in:label,body,title	0	1	1
is:pr error in:label,body,title	49	2448	2497
is:pr flaw in:label,body,title	0	9	9
is:pr failure in:label,body,title	8	468	476

Table 3: Keywords based filtering for pull requests (pr). Own work.

The keywords filtering applied to pull requests produced much better results than the labels one as it can be seen in table 3. The "bug" and "failure" keywords, applied to a broader scope, returned a much higher amount of pull requests that could be identified as bugs fixes. Similarly to the issues, the generic "error" keyword produced many more results and it was therefore given a lower priority in the pull requests filtering process.

The filtering of issues and pull requests from GitHub, the data preparation and the database insertions were all performed through a python script that was developed for this research. This part of the experiment relied on the fact that a MySQL database was provided to store the data. A few steps were needed to produce the final data-set.

The first step included the creation of two database tables to store data for issues and pull requests respectively. The retrieval of data from GitHub was performed using Perceval, a python module that offers dedicated functionalities for this kind of software development analysis [11]. As Perceval uses the GitHub API, the amount of information it returns is quite sizeable [10] and not all of it was needed for this research.

From the data model that Perceval returned, only the following information was kept for issues and pull requests respectively: unique identifiers, common identifiers, title, description, GitHub urls, creation date, closure date, update date, status, labels and the number of comments, commits, additions, deletions and changed files. This was then reflected in the schema of the issues and pull requests tables in the database.

The second step was the actual data retrieval from GitHub using the developed python script that leveraged the Perceval functionalities. The data received for both issues and pull requests was then filtered using the keywords approach described above. Superfluous information was then removed to match the database schema before the data was added to the respective table. In this step, only the issues and the pull requests that were "closed" were taken into account since this research focuses on analyzing bugs with an existing fix. A total of 2539 issues and pull requests were added to their database tables.

The third step produced a unique table that reflected the schema of the issues and pull requests tables and included the information about bugs (from the issues table) and their respective fixes (from the pull requests table). To achieve this, an inner join query was performed between the issues and pull requests tables using the common identifier as the related column [35]. With a small number of entries being bugs that were not linked to a fix, this resulted in a final table of 2400 bugs linked to their fix.

The fourth and last step focused on removing false positives from the data-set which were represented by: entries where no fix occurred (0 as changed files value) and entries with missing title, description and/or labels as they would not provide useful information. This resulted in 2285 bugs with a fix stored in the database. Therefore the goal of creating a data-set with about 2000 entries was achieved.

4.2 Analyzing Bugs

The analysis of the collected bugs for the Moby configuration management system was performed through manual inspection. From the total of 2285 bugs a random sample of 100 was selected to be analyzed. This choice was taken since the manual analysis was a time consuming approach and it was not feasible to go through the whole data-set because of the time constraints of the research.

Building upon the work by Chaliasos et al. from 2021, the bugs analysis was peer reviewed by three other researchers during the process [8]. The task was spilt into five iterations of 20 bugs taken from the random sample of 100 bugs. During these iterations, bugs were assigned categories to label their symptoms, root causes, impact, fixes, system dependency and triggers. At the end of each iteration, the results were peer reviewed, discussed and adjusted if needed. This step was important to minimize the subjectivity aspect of the manual inspection that could have led to biases in the classifications.

The manual analysis of each bug was performed by inspecting their bug reports provided in the GitHub issue tracker. Bug reports are an important part of software development as they document the occurrence of defects alongside their severity and priority of fixing. A clear classification of bugs through their reports can improve efficiency in their detection and prevention as well as the overall software maintainability [24]. The categories that were identified during the bugs analysis are related to each research question and provided the necessary information to answer them. The following categories were defined for each element outlined in the research questions.

RQ1 - Symptoms: A symptom shows how a user understands that something is going wrong in the configuration management system while using it [8].

- **Unexpected Run-time Behavior (URB):** This manifests when the system is executing its tasks, however no crash is involved. These bugs can be related to: containerized applications run-time executions (**Container Image Behavior Error, URBCIBE**); erroneous parsing of the user input (**Configuration Does Not Parse as Expected, URBCDNP**); faulty configuration in the target machine (**Target Misconfiguration, URBTM**).
- **Misleading Report (MR):** This appears when the configuration management system emits a false warning or a false error message. This includes errors and warnings that are not clear enough to understand the bugs origin.
- **Unexpected Dependency Behavior Error (UDBE):** This shows when there is an error related to a dependency of the configuration management system.
- **Performance Issue (PI):** Characterized by the system using too much time and memory resources.
- **Crash (C):** This occurs when the system experiences a crash. It can relate to: feature not functioning due to module/non-core errors (**Feature/Sub-Feature Non Functional, CFNF**); core component task execution (**Execution Crash, CEC**); configuration parsing (**Configuration Parsing Crash, CCP**); environment specific errors related to the underlying operative system (**Environment Related Error, CERE**).

RQ2 - Root Causes: Root causes help identify the reasons of defects or failure events. Once this becomes clear, the bug can be fixed at its source [7].

- **Container Image Life-Cycle Bug (CILB):** This concerns bugs that are pertinent to an image expected life-cycle, from its creation to its demotion.
- **Error Handling & Reporting Bugs (EHRB):** This shows when an error is correctly identified, but its handling and reporting does not produce the expected results.
- **Misconfiguration Inside The Code-Base (MC):** The code-base contains either an incorrect default value configuration (**MCDV**) or dependency configuration (**MCDP**).
- **Target Machine Operations (TMO):** This occurs in the server side of the system and relates to: faulty file-system operations (**TMOFS**); dependencies issues (**TMOD**); fetching values failure (**TMOFTMF**); configuration parsing issues (**TMOPI**); erroneous instructions translation (**TMOITE**).
- **Controller Machine Operations (CMO):** This shows in the controller side of the system and relates to: faulty statements execution (**CMOEP**); connections issues (**CMOCONP**); configuration parsing errors (**CMOPI**).

RQ3 - Impact: The more critical the bug is, the more dangerous the consequences on the software system are [18]. The impact level can affect specific edge cases only (**Low**), prevent the running system to perform some important tasks (**Medium**) or terminate the system operations and prevent it from functioning (**High**).

From a user's perspective the bugs impact can have multiple consequences: container operations crash (**CNTOC**); security hazard (**SH**); performance degradation (**PD**); logs reporting failure (**LOGRTF**); target system configuration

failure (**TCFC**), inaccurate (**TCIA**) or incomplete (**TCIN**).

RQ4 - Fixes: The work by Zhao et al. from 2017 provided a fine grained classification of bug fixing activities based on code changes [38]. On top of this work, the changed entities definitions by Wang et al. from 2018 were also applied [36]. These categories were used and expanded in this research to define the bugs fixes from the analyzed data sample both on a code and conceptual level.

On the code level, the following categories identify the changes that were made to the actual lines of code: change on data declaration (**CDDI**); change on assignment statement (**CAS**); add class (**AC**); remove class (**RC**); change class (**CC**); add method (**AM**); remove method (**RM**); change method (**CM**); invoke method (**IM**); change loop statement (**CLS**); change branch statement (**CBS**); change return statement (**CRS**).

On a conceptual level, the following categories relate to broader areas of the system the fix targeted: fix execution (**FEC**), parser (**FPC**) or connectivity (**FCC**) components; expand execution (**EEF**), parser (**EPF**) or connectivity (**ECF**) features; change dependencies (**CDEP**), system structure (**CSS**) or configuration (**CCONF**); fix diagnostic messages to user (**DDM**).

RQ5 - System Dependency: Moby bugs reports provided information about the underlying operative system and eventually specified if the issue was system related (**Dependent**) or not (**Independent**).

RQ6 - Triggers: A trigger describes how a bug was introduced to the system. This can be related to many factors like human errors or system design errors, for example [8].

- **Logic Errors (LE):** This relates to defects in logic or branching of a procedure.
- **Algorithmic Errors (AE):** This is identified from a wrong implementation of an algorithm or when a wrong algorithm was used.
- **Configuration Errors (CE):** A configuration error is one of the major causes of a system failure. For example, an error in the immutable description of dependencies through file declaration can introduce a failure in every created instance of the system [3]. Configuration parameters, dependencies and components compatibility are the most common errors in this category [37].
- **Programming Errors (PE):** This relates to errors caused by a programmer such as out-of-bounds array accesses or accesses to null references, for example.

Alongside trigger errors, how users can reproduce the bugs was summarized in the following categories: command line inputs (**CLIC**) that can be container (**CLICCC**) or module (**CLICDMO**) specific; environment setup (**ENVS**); faulty dependency usage (**FDEPU**); operative system specific executions (**OSSE**); test cases (**TC**); specific invocation (**SI**) of target machine operations (**SITMCE**), internal modules (**SIIMI**), custom modules (**SICMI**), target machine parsing (**SITMRP**) or configuration parsing (**SICRP**).

5 Experimental Results

The bugs analysis process allowed the classification of the 100 sampled bugs for the Moby configuration management system. From the collected data it was possible to extract the experiment results in line with the research question.

The goal of this section is to provide the answer for the six research questions this research aims to address. Each subsection discusses the results regarding bugs symptoms (5.1), root causes (5.2), impact (5.3), fixes (5.4), system dependency (5.5) and triggers (5.6).

5.1 RQ1 - What are the symptoms of these bugs?

From figure 1 it can be seen that the most common symptoms are related to unexpected run-time behaviors (URB), crashes (C), misleading reports (MR) and unexpected dependency behaviors (UDBE). The majority of URB bugs were linked to container images (URBCIBE) and configuration issues (URBCDNP and URBTM). Crashes occurred more frequently in environment related errors (CERE) compared to execution ones (CEC).

MR bugs were mostly related to ambiguous error messages thrown when an operation failed. This included no

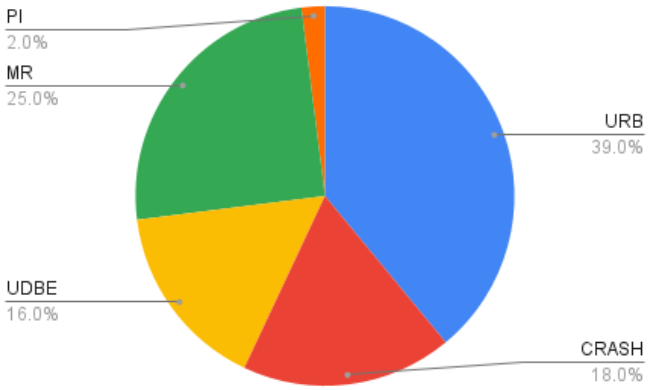

Figure 1: Distribution of symptoms found in the 100 sampled bugs from Moby. Own work.

error message at all, which leaves no useful details about the bug. In terms of UDBE bugs, the modular nature of Moby might be a factor for their prevalence. As Moby relies on dependencies such as *containerd* and *linuxkit* [21], a bug thrown from them will affect Moby operations as well. Regarding the performance issue bugs (PI), the small detected amount did not lead to any relevant conclusion besides that bugs in Moby do not seem to affect its performance.

It is important to underline how Moby can be used to create custom container images [21]. The URBCIBE symptom covers 18% of the bugs sample and shows that a container has been created successfully, but it does not behave as expected. Getting unexpected results from one of the system core features can lead to further problems. It is therefore recommended to strengthen the validation process of created containers with additional testing.

5.2 RQ2 - What are the root causes of these bugs?

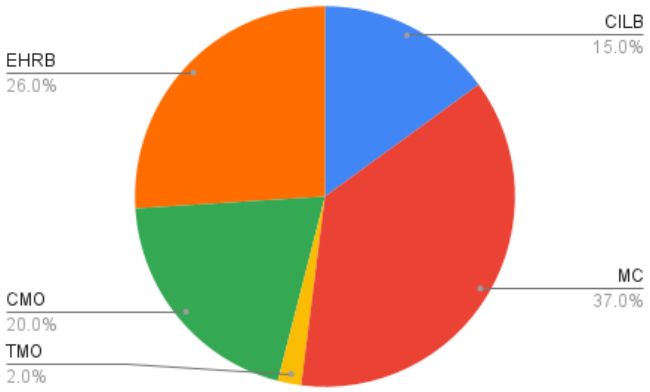

Figure 2: Distribution of root causes found in the 100 sampled bugs from Moby. Own work.

Figure 2 shows that the most common root cause is related to misconfigurations (MC). Misconfiguration of dependencies (MCDP) took the vast majority of this share, which implies that the versioning, import and usage of dependencies in the code-base is faulty. This is in line with what was observed in 5.1 as MCDP bugs can be the root cause of symptoms like UDBE, URB and Crashes.

Additionally, as a consequence of the MR symptoms, error handling (EHRB) is one of the most common root causes of bugs. This means that the reporting of an error is faulty and the system cannot properly address the bug to the user. Root causes linked to controller execution problems (CMO) are also a reflection of of the URB and Crashes symptoms as they represent errors occurring when performing tasks. Similar considerations can be applied to the relation between URBCIBE symptoms and container image life-cycle (CILB) root causes as they both highlight problems about containers. Lastly, the low detection of the target machine operation (TMO) root cause did not lead to any relevant conclusion.

5.3 RQ3 - What is their impact?

Figure 3 shows how most of the analyzed bugs are critical to the correct functioning of system. This is very likely due to the granular nature of Moby, which relies on the correct functioning of all its modules to perform flawlessly as a whole.

From the bugs, the user experiences mostly target operation failures (TCFC), which leave him/her unable to use Moby to complete a task. Logs reporting failure (LOGRTF) is also a relevant consequence as it relates to any error that does not report information correctly leaving the user with little knowledge about the issue. The occurrence of container crashes (CNTOC) continues the pattern noticed with symptoms and root causes about issues in one of Moby's core features. Reports found about experiencing performance degradation (PD) and security hazards (SH) were included in the results but did not lead to relevant conclusions.

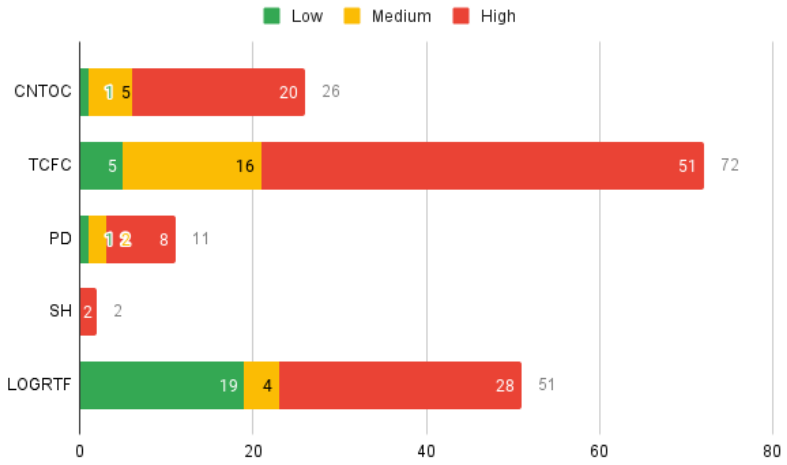

Figure 3: Distribution of bugs consequences in the 100 sampled bugs from Moby related to their impact. Own work.

5.4 RQ4 - How do developers fix these bugs?

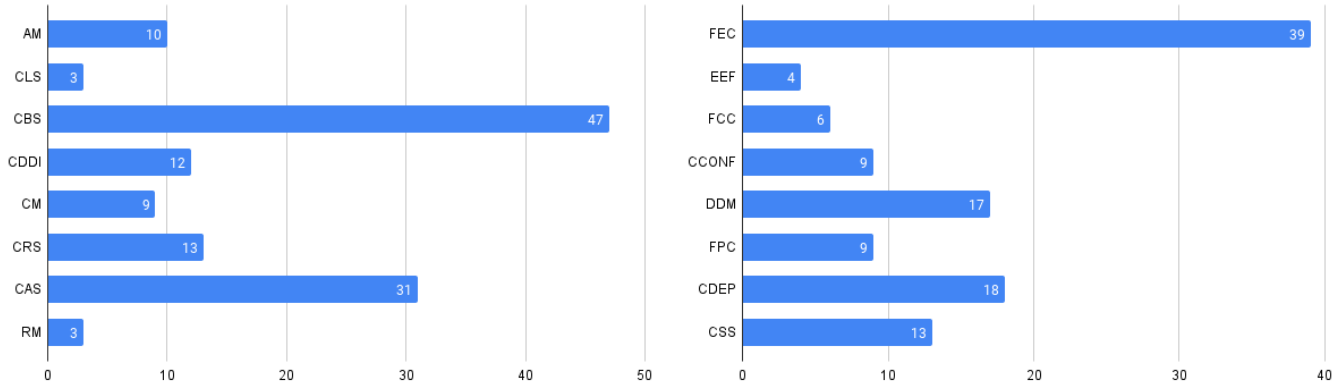

Figure 5: **Left:** Distribution of code fixes in the 100 sampled bugs from Moby. **Right:** Distribution of conceptual fixes in the 100 sampled bugs from Moby. Own work.

As the left side of figure 5 shows, the majority of the fixes in Moby involve small changes in terms of lines of code. This is reflected by the high number of changes on branch statements and assignment statements. On the right side of figure 5 it can be seen how the fixes were applied on a conceptual level on specific areas of Moby. Most of the fixes aimed to correct errors in execution components, which is in line with how users mostly experience bugs as operation failures (5.3). The relevant amount of changes in dependencies and in diagnostic messages is also in line with what was discussed in 5.2 and 5.3.

5.5 RQ5 - Are these bugs system dependent?

Bugs in Moby resulted in a slight majority being system independent as can be seen from the left side of figure 7. The right side of the figure shows the distribution of the operative systems in the analyzed bugs. It can be observed how Linux is the system with the largest percentage of reported bugs compared to Windows and MacOS, however bug reports usually listed Linux generically without specifying the distribution. Therefore only bugs with explicit mention of an operative system version are likely related to it.

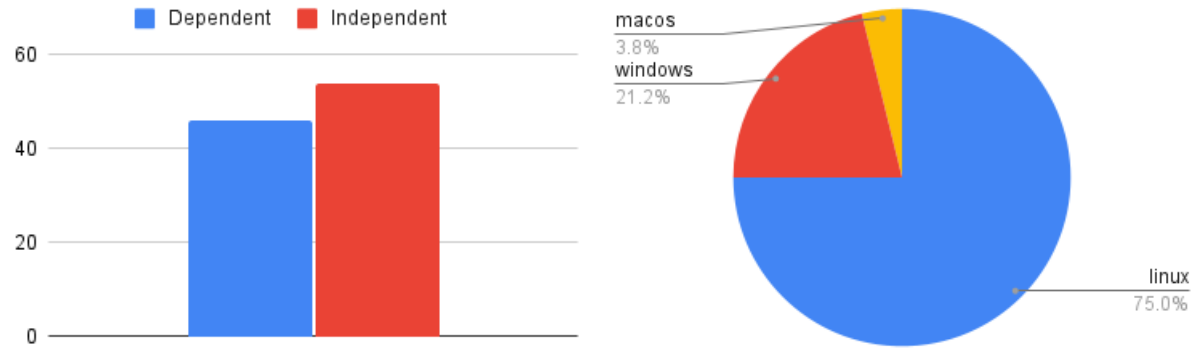

Figure 7: **Left:** Distribution of system dependency in the 100 sampled bugs from Moby. **Right:** Distribution of operative system dependency in the 100 sampled bugs from Moby. Own work.

5.6 RQ6 - What triggers these bugs?

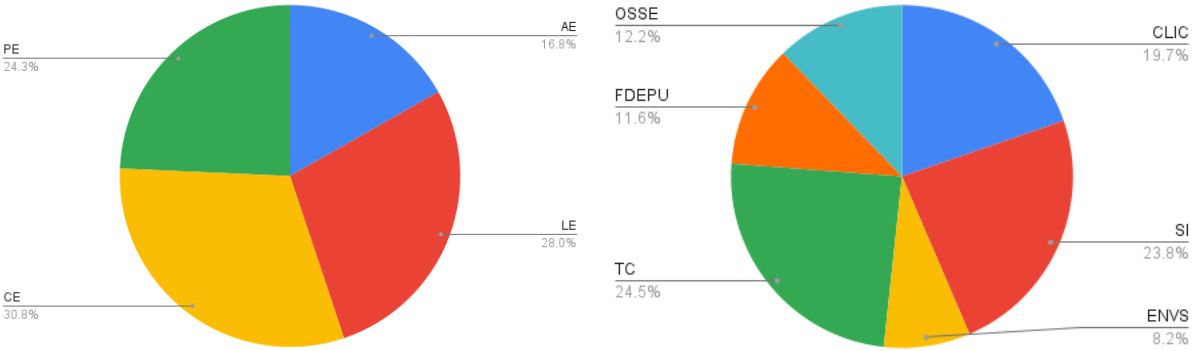

Figure 9: **Left:** Distribution of error triggers in the 100 sampled bugs from Moby. **Right:** Distribution of bugs reproductions in the 100 sampled bugs from Moby. Own work.

The left side of figure 9 shows how the triggers distribution is quite balanced across the categories. Configuration errors (CE) and logic errors (LE) represent the most common bugs triggers. This is in line with the results obtained for the bugs fixes (5.4). Programming errors (PE) and algorithmic errors (AE) were also considered in line with previous findings about the bugs fixes.

On the other hand, bugs reproducibility in Moby is quite diverse as the right side of figure 9 illustrates. Test cases (TC) execution can be found in only 24.5% of the analyzed bugs, meaning that fixes were mostly not added together with a proper check. Other relevant characteristics are command line operations (CLIC), such as container specific command (CLICCC) and module specific execution (CLICDMO). Specific invocations (SI), configuration of dependencies (FDEPU) and executions in specific operative systems (OSSE) are also large percentages of the results.

The diversity in how bugs are reproduced can be seen as a consequence of the lack of test cases provided with the fixes. This means that users need to follow specific setups and instructions in order to be in the position to trigger the bug again, which is much less convenient than just running the Moby test suite.

6 Discussion

As the analysis and characterization of bugs in the Moby configuration management system is a very specific goal, it is also important to contextualize this study in a broader perspective. The aim of this section is to discuss the insights and main takeaways from the results of this research (subsection 6.1) as well as the threats to the validity of its methodology and results (subsection 6.2).

6.1 Insights and Main Takeaways

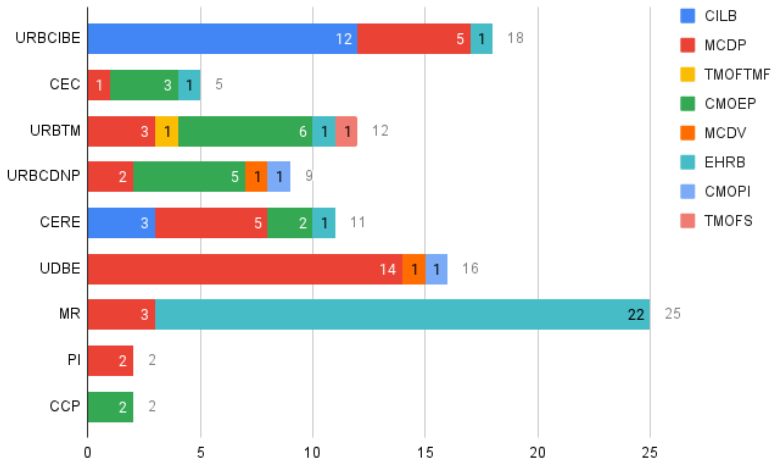

Figure 10: Distribution of bugs symptoms in the 100 sampled bugs from Moby related to their root causes. Own work.

Interpolating symptoms and root causes gives an overview of how each bug symptom manifests itself. It can be observed from figure 10 that each symptom tends to have a major root cause related to it, therefore bugs usually appear following some patterns. This can support developers and researchers with the study and implementation of an additional dedicated test suite that Moby should run once a fix is submitted as a pull request. This test suite should be able to generate enough cases to support the detection and prevention of bugs. For example, an URBCIBE bug could be prevented by testing many containers actions and configuration settings and comparing them with the expected results.

As the right side of figure 9 reported, only 24.5% of fixes included a test case that could trigger the bug, which means that reproducing bugs in Moby can be considered quite difficult overall. This is not an ideal setup for contributors to the Moby open source repository since it does not allow them to test fixes properly. On top of this, a fix could also cause another bug that might not be triggered by the existing test suite before the changes are approved and merged in the main branch. It is recommended to the Moby main maintainers to introduce stricter approval policies for issues and pull requests, including: (1) presence of an existing issue that clearly describes the bug and how it was triggered; (2) use of labels identifiers for bugs and bugs fixes; (3) presence of a structured description of the pull request, including reference to the solved issue and how the fix was performed; (4) fixes must come with test cases so the reviewers can trigger the bug easily to approve the changes.

Finally, as Moby is an open source project that went through significant refactoring through its lifetime [21], it is important to avoid the same bugs pattern repetition at every big change. It is recommended to researchers to perform a study on the history of bugs in Moby and develop a tool that shows how bugs appeared during the project lifetime and which changes led to them. This will support the developers of Moby with preventing recurring bug patterns to appear at every significant refactoring.

6.2 Threats to Validity

The validity of the research can be challenged by several considerations which represent threats of internal, external and construct validity [34]. The manual analysis approach also could introduce doubts about the objectivity of the reported results as the bugs classification was subject to the judgement of the researcher.

A potential threat to internal validity is the collection criteria used for the bugs in the data gathering phase. As the predefined labels filtering produced disappointing results, a keywords based approach was chosen instead. This method is challenged by Chaliasos et al. in their work from 2021, where it is stated that a keywords based search can lead to missing relevant bugs [8]. However, the amount of results gathered in the current research using the keywords based filtering was in line with the initial requirement of 2000 bugs. On top of this, during the analysis of the bugs sample, the small number of false positives found supported the case that the majority of the retrieved bugs was relevant.

External validity can be undermined by the potential lack of representation of the bugs selected for the analysis phase. The choice of picking a random sample was in line with the previous bugs studies by Zhao et al. from 2017 [38], Wang et al. from 2018 [36] and Chaliasos et al. from 2021 [8]. However, it is important to consider that Moby went through a significant transition when the Docker team made the project open source in 2017 [21]. This means that the distribution of bugs in the period that goes from 2017 to the time this research was performed could also be dependent on that. As the research did not focus on the timeline underlying the different bugs, these considerations did not impact the results. That being said, taking the time aspect into account can provide interesting material to researches about the evolution of bugs in Moby.

The construct validity can be challenged by the subjectivity that the manual analysis of bugs could introduce. In order to keep the results as objective as possible, each round of categorization was reviewed by other researchers. The feedback gathered from other point of views was used to refine the categories and the classification of bugs. This approach was in line with the methodology proposed by Chaliasos et al. in 2021 [8] and by Wang et al. in 2018 [36].

7 Related Work

The lack of previous literature about bugs analysis in configuration management systems led to the necessity of finding bugs studies that focused on other areas [1] [20]. This section aims to discuss the most relevant works found to support the research: Zhao et al. from 2017 [38], Wang et al. from 2018 [36] and Chaliasos et al. from 2021 [8].

The most similar work to this research in terms of methodology is the study by Chaliasos et al. from 2021, which focuses on typing-related bugs in JVM compilers. The symptoms, root causes, fixes and triggers of bugs were defined by studying a random sample of 320 bugs from Java, Scala, Kotlin and Groovy. The key findings of this work highlighted the following: (1) compile time errors and performance degradation are common symptoms for typing-related bugs; (2) most of root causes relate to correctness issues in the core components of the compilers; (3) non-compilable test cases contribute to causing bugs; (4) invalid programs are the most common trigger for typing-related bugs [8].

Although this research focuses on the Moby configuration management system, the methodology proposed by Chaliasos et al. provided a strong starting point and contributed significantly to the project setup. A key difference was the choice of interpretation of bugs fixes. Chaliasos et al. analyze the fixes in terms of how the bugs are introduced and what is the size of the changes [8]. In this research, the size of the fixes was considered together with a more fined grained categorization that focused on the actual changes in the source code and project structure.

The work by Zhao et al. from 2017 focuses on change types in bug fixing code and provides a taxonomy to help classify code changes. The following results were found: (1) interface related code is the most frequently changed by bug fixes; (2) changes on function calls and branch statements are the most frequent; (3) changes on common sub-types have similar trends [38]. The categorization of code changes based on data, computation, interface and logic control was applied to the current research to better identify the fixes of bugs in Moby. The main difference was the interpretation of interface related categories. Zhao et al. analyze interfaces as functions related faults based on their definitions and calls [38]. In the current research, the interface category was subdivided using more specific subcategories related to both functions and classes.

The work by Wang et al. from 2018 provides an empirical study of multi-entity changes in bug fixes by analyzing their frequency, content and semantic meanings. The key findings of this work suggest that: (1) 66% to 76% of changed entities in bug fixes are semantically related; (2) the most frequent bug fix is a change in a method definition; (3) different fixes may still apply similar contents and method invocations [36]. The definition of a program entity as a class or method alongside the fixes categories were adopted in the current research. This allowed a more detailed categorization of the Moby interfaces bug fixes.

8 Responsible Research

Research integrity is based on principles that guide the parties involved in the research, all of which are expected to follow responsible research practices [23]. This section aims to discuss the ethical aspects of the conducted research, the main responsible actions taken in the data gathering and the steps taken to ensure that the work is reproducible.

The study of bugs in Moby involved the processing of open source information from GitHub. The bugs and fixes database and the results of the bugs analysis only contained information relative to the source code of Moby. However, the data retrieved as a whole from the GitHub API through the Perceval module included information about the user that opened/closed the issue or pull request [10].

As the user details were not part of the scope for this research, the choice was to avoid their retrieval from the beginning. This was done through the "–filter-classified" functionality of the Perceval module, which omits all classified information from the returned data [9]. As this research aims to provide a methodology that can be reused and improved by other parties, the protection of user information has been a priority during this work.

The process of gathering bugs and analyzing them presented some disappointing results along the way which needed to be understood and eventually improved. For example, the first iteration of labels based bugs gathering from GitHub produced a much smaller number of results than anticipated. However, the keywords based approach provided the number of results necessary, which has been carefully detailed in section 4. Full transparency in research is fundamental and negative outcomes must be presented as well [23]. A study by Fanelli in 2012 highlighted how more than 85% of published studies claimed to have reached positive results, which raised doubts about scientific objectivity [17].

The reporting of the full process and the full results obtained during this research has been very important. Each negative outcome was viewed as a key part of the process that allowed the research to achieve its goals as well as providing more ground for further improvement. As future work can be built upon this research, providing both positive and negative outcomes will enhance reproducibility. This is underlined in a study by Mehta in 2019, which recommended that researchers must recognize all important work, irrespective of its outcome [27].

Reproducibility drives progress through different generations of scientists as they will be able to build on previous achievements [2]. A potential reproducibility crisis was discussed by Baker, whose 2016 study highlighted how 70% of researchers failed to reproduce experiments from others, and more than 50% failed to reproduce their own [6]. As computation became a key part of research, it also became less obvious how to reproduce the work based on the text of papers and articles only. It is therefore important to respond to the potential reproducibility crisis by making all details of the research computations easily available to other researchers [32].

This research was conducted with the goal to be reproducible and serve as ground for future work. The followed methodology was explained in detail and broken down to each step, where both negative and positive outcomes have been reported and explained. As data and computations represented a key part of the work, all the gathered bugs and developed scripts have been made available through open source platforms. The bug entries and the analyzed sample with the results are hosted on a public MySQL database, while the python scripts used for the computation are available on GitHub alongside a detailed documentation.

As this research followed responsible research practices it is hoped that it will encourage and facilitate future work to apply the research integrity principles and strive for transparency and reproducibility.

9 Conclusions and Future Work

The research aimed to analyze and characterize previously reported bugs in the Moby configuration management system. To achieve this, bugs were categorized based on their symptoms, root causes, impact, fixes, system dependency and triggers. The necessary bugs were collected through a filtering process of the Moby GitHub issues and pull requests. On top of labels identifiers, specific keywords were used to avoid the missing of relevant data.

The categorization process focused on a random sample of 100 bugs taken from the collected ones and it was peer reviewed by fellow researchers to reduce subjectivity as much as possible. The findings suggested that the most common symptoms for bugs in Moby are related to misleading errors reporting, containerized applications behavior and errors from dependency modules. Root causes of bugs are mostly linked to faulty dependencies configuration, error

handling and reporting, container images operations and Moby core component run-time execution. The impact that bugs have on the system leads to wrong configuration of target machines and containerized applications as well as problems in the logging mechanism. As Moby is a modular framework, bugs that appear in a module and compromise its correct functioning will also impact the whole system as well. This is why the majority of Moby bugs can be labelled as highly critical. The majority of bugs fixes in Moby involve small changes to the code including changes on branch statements, value assignments, return statements and data initialization. The bugs were split between system dependent and system independent, and most were identified in Linux although this might be misleading as no specific distribution was specified in the report in most cases. Finally, bugs tend to be triggered by logic, configuration, programming and algorithmic errors. The reproducibility of the bugs is mostly related to command line operations and specific modules execution.

The results from the research highlighted interesting main takeaways about Moby bugs and their behavior. Each identified symptom is mostly linked to a specific root cause, which means that bugs usually appear following a pattern. This can be used as a starting point to study and develop a dedicated test suite that can support the detection and prevention of the most common bugs. On top of this, bugs in Moby are quite difficult to reproduce as only a small percentage of the fixes included a test case. A more structured workflow for reporting and fixing bugs can improve this problem. This includes adding detailed bugs report to GitHub issues, properly linking pull requests to issues and setting the requirement of providing test cases alongside fixes.

The methodology proposed by this research about studying bugs in configuration management systems can be further expanded and improved. The keywords based filtering provided the data needed for the analysis and resulted in a small number of false positives. However this approach can become more sophisticated by training algorithms that look at the bug reports as a whole rather than only looking at specific terminology. Another interesting addition to the bugs analysis can be the study of the evolution of bugs over time. As projects can go through significant changes, analyzing which bugs occur after major refactoring can help maintainers prevent them.

This research fulfilled its aim of providing an analysis and characterization of previously reported bugs in the Moby configuration management system. On top of this, a methodology on which further work can be built was proposed as well as insights that can support developers in identifying bugs. As literature on configuration management system focused so far on performance rather than failures, this work provides an additional perspective in the field.

Acknowledgements

Many thanks to fellow researchers Mykolas Krupauskas, Bryan He and Matas Rastenis for their peer review of the bugs analysis. Responsible professor prof. dr. ir. Diomidis Spinellis and supervisor Thodoris Sotiropoulos are also thanked for their supervision, advice and for making this research possible together with the Delft University of Technology.

References

[1] Configuration management process capabilities. *Procedia CIRP*, 11:169–172, 2013. ISSN 2212-8271. doi: https://doi.org/10.1016/j.procir.2013.07.043. 2nd International Through-life Engineering Services Conference.

[2] David B. Allison, Richard M. Shiffrin, and Victoria Stodden. Reproducibility of research: Issues and proposed remedies. *Proceedings of the National Academy of Sciences*, 115(11):2561–2562, 2018. doi: 10.1073/pnas.1802324115.

[3] Ahmed Amamou, Martin Camey, Christophe Cerin, Jonathan Rivalan, and Julien Sopena. Resources management for controlling dynamic loads in clouds environments. the wolphin project experience, 2020. URL https://hal.archives-ouvertes.fr/hal-02481264.

[4] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults in web api integration in a large-scale payment company. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 13–22, United States, 2018. Association for Computing Machinery (ACM). doi: 10.1145/3183519.3183537. ICSE 2018 : 40th International Conference on Software Engineering ; Conference date: 27-05-2018 Through 03-06-2018.

[5] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, api usages, and differences. *Proceedings of the ACM on Programming Languages*, 4:1–32, 11 . ISSN 2475-1421. doi: 10.1145/3428282.

[6] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533:452–454, 5 . ISSN 0028-0836. doi: 10.1038/533452a.

[7] Stephen J. Bigelow. How to handle root cause analysis of software defects, 2021. URL https://www.techtarget.com/searchsoftwarequality/tip/How-to-handle-root-cause-analysis-of-software-defects.

[8] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. Well-typed programs can go wrong: a study of typing-related bugs in jvm compilers. *Proceedings of the ACM on Programming Languages*, 5:1–30, 10 . ISSN 2475-1421. doi: 10.1145/3485500.

[9] CHAOSS. Classified fields in perceval docs should correspond to the fields actually removed, 2020. URL https://github.com/chaoss/grimoirelab-perceval/issues/611.

[10] CHAOSS. Retrieving data from github repositories, 2021. URL https://perceval.readthedocs.io/en/latest/perceval/github.html.

[11] CHAOSS. Perceval, 2021. URL https://perceval.readthedocs.io/en/latest/perceval/intro.html.

[12] D. Chhillar and K. Sharma. Act testbot and 4s quality metrics in xaas framework. pages 503–509. ISBN 9781728102115. doi: 10.1109/COMITCon.2019.8862212.

[13] TU Delft. Search operators, 2022. URL https://tulib.tudelft.nl/searching-resources/search-operators/.

[14] TU Delft. Making a search plan, 2022. URL https://tulib.tudelft.nl/searching-resources/making-a-search-plan/.

[15] Anthony Di Franco, Hui Guo, and Cindy Rubio-GonzÃ¡lez. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi: 10.1109/ASE.2017.8115662.

[16] Ahmet Efe, Ulaay Aslan, and Aytekin Mutlu Kara. Securing vulnerabilities in docker images. *International Journal of Innovative Engineering Applications*, 4:31–39, 6 . ISSN 2587-1943. doi: 10.46460/ijiea.617181.

[17] Daniele Fanelli. Negative results are disappearing from most disciplines and countries. *Scientometrics*, 90:891–904, 03 2012. doi: 10.1007/s11192-011-0494-7.

[18] Thomas Hamilton. Severity priority in testing: Differences example, 2022. URL https://www.guru99.com/defect-severity-in-software-testing.html.

[19] Richard Hatheway. Why enterprise it organizations will benefit from application containerization, 2021. URL https://www.cio.com/article/189567/why-enterprise-it-organizations-will-benefit-from-application-containerization.html.

[20] Johannes Hintsch, Carsten Goerling, and Klaus Turowski. A review of the literature on configuration management tools. 05 2016. URL https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1005&context=confirm2016.

[21] Solomon Hykes. Introducing moby project: a new open-source project to advance the software containerization movement, 2017. URL https://www.docker.com/blog/introducing-the-moby-project/.

[22] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. pages 77–88. ACM, 6 . ISBN 9781450312059. doi: 10.1145/2254064.2254075.

[23] KNAW, NFU, NWO, TO2-federatie, and Vereniging Hogescholen. Netherlands code of conduct for research integrity.

[24] D.-G. Lee and Y.-S. Seo. Improving bug report triage performance using artificial intelligence based document generation model. *Human-centric Computing and Information Sciences*, 10, 2020. doi: 10.1186/s13673-020-00229-7.

[25] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc. pages 517–530. ACM, 3 . ISBN 9781450340915. doi: 10.1145/2872362.2872374.

[26] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security. pages 418–429. ACM, 12 . ISBN 9781450365697. doi: 10.1145/3274694.3274720.

[27] Devang Mehta. Highlight negative results to improve science. *Nature*, 10 . ISSN 0028-0836. doi: 10.1038/d41586-019-02960-3.

[28] Moby. Moby project, 2017. URL https://mobyproject.org/.

[29] Moby. Moby project, 2022. URL https://github.com/moby/moby.

[30] Kaushik Sen. What is configuration management and why is it important?, 2022. URL https://www.upguard.com/blog/5-configuration-management-boss.

[31] SPK. How bug analysis improves software engineering postmortems, 2011. URL https://www.spkaa.com/blog/how-bug-analysis-improves-software-engineering-postmortems.

[32] V.C. Stodden. Reproducible research. *Computing in Science & Engineering*, 12:8–13, 9 . ISSN 1521-9615. doi: 10.1109/MCSE.2010.113.

[33] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. A comprehensive feature comparison study of open-source container orchestration frameworks. *Applied Sciences (Switzerland)*, 9, 2019. ISSN 20763417. doi: 10.3390/app9050931.

[34] Duquesne University. Overview of threats to the validity of research findings, 2022. URL http://www.mathcs.duq.edu/~packer/Courses/Psy624/Validity.html.

[35] W3School. Mysql joins, 2022. URL https://www.w3schools.com/mysql/mysql_join.asp.

[36] Ye Wang, Na Meng, and Hao Zhong. An empirical study of multi-entity changes in real bug fixes. pages 287–298. IEEE, 9 . ISBN 978-1-5386-7870-1. doi: 10.1109/ICSME.2018.00038.

[37] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. 2011. URL https://www.sigops.org/s/conferences/sosp/2011/current/2011-Cascais/printable/12-yin.pdf.

[38] Y. Zhao, H. Leung, Y. Yang, Y. Zhou, and B. Xu. Towards an understanding of change types in bug fixing code. *Information and Software Technology*, 86:37–53. doi: 10.1016/j.infsof.2017.02.003.