The Evolution of the Hadoop Distributed File System

Stathis Maneas, Bianca Schroeder
Department of Computer Science
University of Toronto,
Email: *smaneas@cs.toronto.edu*, *bianca@cs.toronto.edu*

***Abstract*—Frameworks for large-scale distributed data processing, such as the Hadoop ecosystem, are at the core of the big data revolution we have experienced over the last decade. In this paper, we conduct an extensive study of the Hadoop Distributed File System (HDFS)'s code evolution. Our study is based on the reports and patch files (patches) available from the official Apache issue tracker (JIRA) and our goal was to make complete use of the entire history of HDFS at the time and the richness of the available data. The purpose of our study is to assist developers in improving the design of similar systems and implementing more solid systems in general. In contrast to prior work, our study covers all reports that have been submitted over HDFS's lifetime, rather than a sampled subset. Additionally, we include all associated patch files that have been verified by the developers of the system and classify the root causes of issues at a finer granularity than prior work, by manually inspecting all 3302 reports over the first nine years, based on a two-level classification scheme that we developed. This allows us to present a different perspective of HDFS, including a focus on the system's evolution over time, as well as a detailed analysis of characteristics that have not been previously studied in detail. These include, for example, the scope and complexity of issues in terms of the size of the patch that fixes it and number of files it affects, the time it takes before an issue is exposed, the time it takes to resolve an issue and how these vary over time. Our results indicate that bug reports constitute the most dominant type, having a continuously increasing rate over time. Moreover, the overall scope and complexity of reports and patch files remain surprisingly stable throughout HDFS' lifetime, despite the significant growth the code base experiences over time. Finally, as part of our work, we created a detailed database that includes all reports and patches, along with the key characteristics we extracted.**

***Index Terms*—Distributed File Systems, Open-Source, Code Evolution, System Analysis, Hadoop File System (HDFS)**

I. Introduction

Frameworks for large-scale distributed data processing, such as the Hadoop ecosystem [1], are the key enabling technology for the big data revolution of the last decade. In the 10 years since its creation as an open source project, Hadoop has developed into probably the most widespread software framework for distributed data storage and processing: more than half of the Fortune 50 companies use it [2]. One of the core components of such frameworks is the underlying distributed file system, which is responsible for storing the data for fast and reliable distributed access [3]. As a result, a large body of work has been devoted to the various design choices and features of distributed file systems, including for example cache consistency, correctness properties, load balancing, fault detection and reliability, and detailed experimental evaluations of different choices [4], [5], [6], [7], [8].

In this paper, we take a different view of distributed file systems. Rather than focusing on individual features and characteristics, we are more generally interested in the software development process and specifically, how large open-source projects that experience a rapid growth in popularity develop over time and what the main challenges in their development are. The purpose of our study is to provide insight into the evolution of distributed file systems and assist developers in improving the design of similar systems.

Our work focuses on the Hadoop Distributed File System (HDFS) [9], [10] and provides an extensive study of its code evolution. We choose to conduct our study on HDFS, because it is an open-source project, is largely deployed in real-world systems and finally, is under constant development. Our study is based on the reports and patch files available from the official Apache issue tracker (JIRA) instance associated with HDFS. In contrast to existing studies, our goal was to make complete use of the entire history of HDFS and the richness of the data that is available. In this study, we make use of **all** reports that have been submitted over the first nine years of HDFS's lifetime rather than a sampled subset, along with their associated patch files that have been verified by the developers of the system. Additionally, we classify the root causes of issues at a finer granularity than prior work, using a two-level classification scheme that we developed. This allows us to present a different perspective of HDFS, including a focus on the system's evolution over time, as well as a detailed analysis of characteristics that have not been previously studied, such as the scope and complexity of issues in terms of the size of the patch that fixes it and number of files it affects, the time it takes before an issue is exposed, the time it takes to resolve an issue and how these vary over time.

To facilitate our study, a significant amount of work was required in inspecting and manually labelling all 3302 *closed* and *resolved* reports and their corresponding patch files (patches) from the JIRA instance associated with HDFS, covering every released version over the first nine years (2008-2016)[1]. We have stored the results into a database, called *HDFS-DB*, which contains our classification *categories*

[1]We exclude year 2007 from our results because it contains only one report.

and *sub-categories* for all 3302 analyzed reports, along with several graph utilities. *HDFS-DB* can assist existing bug finding and testing tools, by filtering reports based on their assigned *categories*, enabling different types of additional analyses and empirical studies, such as natural language text classification [11], [12].

The analysis of the data in our *HDFS-DB* database led to a number of interesting findings:

- *Bug* reports are the most dominant type among all reports, accounting for more than 50%, followed by *Improvement* reports, which make up about a quarter of all reports and aim to enhance HDFS with additional features.
- The rate of bug reports continuously increases over time; this contradicts the traditional bathtub model for software reliability [13], which assumes relatively stable rates of detected bugs, except for some initial infant mortality and occasional spikes after new releases. We attribute this observation to the highly dynamic, constantly evolving nature of HDFS, whose size and user base have grown tremendously over a relatively short period of time, compared to traditional software projects.
- Compared to previous work, we find a significantly smaller fraction of bugs attributed to memory. For HDFS, 7.2% of all bugs are related to memory, probably indicating that bug detection tools are becoming better at identifying these issues and/or that developers are making heavier use of them.
- We observe that the absolute frequency of concurrency bugs is increasing over time, which is different from what prior work [11] observed for two other open source projects (Mozilla and Apache Web Server). The reason might be that HDFS is being deployed in large, highly-distributed installations, whose scale is continuously increasing over time, thereby creating a stream of new concurrency challenges.
- When looking at the bugs that are detected during an alpha or beta release versus a stable release, we see significant differences. For example, while 60% of all bugs related to incorrect or missing error codes are detected while still in alpha/beta release, only 16% of *null* pointer issues and 19% of performance issues are identified in alpha/beta releases.
- The size of a patch (in kilobytes) and the number of files affected are highly variable, with most patches being small in size, but the largest ones exceeding typical sizes by orders of magnitudes, touching on hundreds of files.

II. Methodology

In this section, we first explain our rationale for choosing to perform our study on HDFS, then we describe our methodology in detail, including information about the official issue repository and our database, and finally, we present our classification scheme in detail.

A. Why the Hadoop Distributed File System (HDFS)

The evolution of distributed processing over the years has provided applications the ability to store and process large data sets in a very efficient way. To this end we, first of all, choose to focus our study on HDFS because it is a great example of a state-of-the-art distributed system, which is widely deployed and used by several different applications to store large data sets. Second, HDFS provides the necessary infrastructure to support the implementation of many other systems. For instance, MapReduce [14], the computation platform of the Hadoop [1] framework, is build on top of HDFS and exploits the functionality provided by the file system. HBase [15] is an open-source, distributed database that runs on top of HDFS, providing the ability to host very large tables. Finally, HDFS has been under constant development and maintenance for more than a decade. The developers of the system continue to resolve emerging bugs and also, to expand the system with new functionality. As a result, the system matures over the years, improving its stability and reliability properties.

B. Data Set Description

The current analysis is based on the HDFS instance of the JIRA [16] issue tracker web repository and covers **all** 55 initial versions, starting from version '0.20.1' released on Sep. 1st, 2009, until version '2.6.5' released on Oct. 08, 2016. The complete release cycle for Apache Hadoop can be found in [17]. For each individual version, we manually process its corresponding released reports. Out of all released reports, we study a total of 3302 reports marked either as *closed*, or as *resolved* and also contain at *least one* patch file. The complete change-log for HDFS can be found in [18].

Based on the provided web repository, for each report, we **automatically** extract and store its *name*, *title*, *type*, *priority level*, *affected components* and *versions*, total *patches*, *commits*, and *comments*, *creation date*, *time to resolve*, *assigned developer* and finally, its *URL*. In addition, we **automatically** track and download all patch files associated with each individual report. As older versions may be included for one patch in the same report, we choose to download and store only its latest version, together with its size (in bytes). To track the latest version, we make use of the information provided by JIRA, which provides the option to mark previous uploaded patch files as *obsolete*. However, this option is not always accurate, since there are reports where the corresponding developers do not make use of the aforementioned option. We discuss this limitation in detail in Section IV. Once the source code of a patch file is downloaded, we calculate its size (in bytes). The total number of downloaded patch files equals to 8280.

In order to facilitate further research in this area, we have stored the results into a a PostgreSQL [19] database, called *HDFS-DB*, which contains our classification *categories* and *sub-categories*, along with several graph utilities. *HDFS-DB* can assist existing bug finding and testing tools, by filtering reports based on their assigned *categories*, enabling different types of additional analyses and empirical studies, such as natural language text classification [11], [12]. Furthermore, our database can be used in order to perform both quantitative and qualitative analyses of HDFS-related issues. Finally, as future work, we plan to complete our classification of the remaining released versions, automate the classification procedure and also expand it to other Apache sub-systems.

C. Classification Scheme Description

In JIRA, by *default*, each downloaded report is assigned two distinct labels, which denote the report's *type* and *priority* respectively. Based on their *type*, reports are *by default* classified into seven different categories, namely *Bug* (issues causing unexpected behaviour), *Improvement* (new features for increased performance, code maintenance, etc.), *Sub-task* (minor tasks related to other reports), *Task* (minor issues to be resolved), *New Feature* (new features that have yet to be developed), *Test* (unit or system tests), and *Wish*.

Based on their *priority*, reports are *by default* classified into five different categories, namely *Major* (important issues related to the system's functionality), *Minor* (minor loss of functionality), *Blocker* (must be resolved immediately), *Critical* (crashes, data loss, memory leaks), and *Trivial* (mostly related to misspelled words, or misaligned text).

In addition to the *default* categorization provided by JIRA, we also develop a two-level classification scheme, where we manually inspect and group reports together into different *categories*, based on the different root causes that triggered them. Our first-level classification procedure makes use of different *categories* per *type* to capture the nature of all the different issues that may arise during the development of HDFS. During our second-level classification procedure, we manually assign an additional *sub-category* to each report, which highlights a particular problem associated with the first-level *category*. For instance, a bug report can involve an issue related to improper memory management (*category*) caused by a resource leak (*sub-category*). We describe the results of our two-level classification scheme in Sections V and VI.

D. Threats to Validity

Our study focuses on one particular system and it is unclear whether our results and observations generalize to other systems. Moreover, the file system used in our study (HDFS) is open-source and thus, our results are not necessarily representative of commercial file systems.

To minimize the possibility of errors during the manual classification, each issue is reviewed twice. In case an issue is complicated or an ambiguity arises, the final classification is performed in such a way until consensus is reached.

For every report in our dataset, JIRA only provides its creation and resolve date; however, developers tend to work on each patch over a possibly discontinuous time interval. Thus, our results do not contain any trends about the work patterns of development within each interval.

Finally, in JIRA, a report contains multiple patch files when the issue affects more than one development or released branches. In this case, individual patches are uploaded for each branch. Moreover, many reports could not be resolved by the first attempted patch; either the issue was not fully resolved or the patch was rejected. A patch file is rejected when it does not comply with the build rules of HDFS (generates documentation warnings or errors in test cases). In case a patch is rejected, the developer continues uploading new versions, until the patch is finally accepted. However, in several reports, the assigned developer does not take advantage of a feature provided by JIRA to mark any previous versions as *obsolete*. As a result, our dataset also contains older patch versions for some reports, since we cannot automatically detect *obsolete* versions. Fortunately, this limitation bias our current aggregate results only by a small factor, since the differences between patch versions are minor and the number of such reports is small.

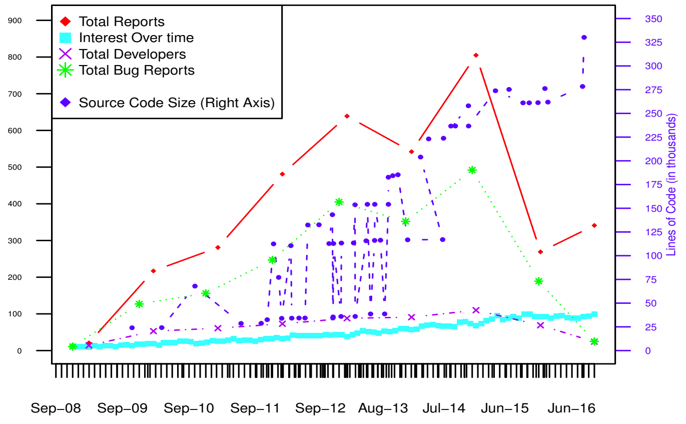


Fig. 1: *Temporal trends of some basic HDFS characteristics.*

III. Frequency and Types of Issues

In this section, we present some high-level findings across the entire database of reports. Specifically, in Section III-A we examine how the number of reports changes over time. In Sections III-B and III-C, we describe the classification results of the default scheme provided by JIRA, based on the specified *type* and *priority*. Finally, Section III-D distinguishes between alpha/beta and stable releases, examining what fraction of issues is reported in each release.

A. Frequency of issues over time

We begin by looking at the overall frequency of issues over time. Figure 1 shows the total number of reports that were entered into JIRA's HDFS archive per year (red line) since the first release in 2008. We note a steep increase in the number of reports over time, beginning with only around 200 reports in 2009 to more than 800 reports in 2015. In addition to all reports, we also look at only those reports related to bugs (dashed green line) where we see a similar trend.

We observe that this continuously increasing number of bug reports differs from the traditional bathtub model commonly cited in software reliability, which assumes a relatively stable bug frequency over time, with the exception of some infant mortality in the beginning and occasional spikes after new releases [13].

There are several possible reasons for this non-traditional increasing trend, most of them related to the highly dynamic nature of the Hadoop ecosystem. As HDFS has developed over the years, its code base has significantly increased in size, from merely 24,035 lines of code (LOC) in 2009 to more than 300,000 LOC in 2016 (blue line), increasing the code complexity and also likely, the bug propensity[2]. Furthermore, over the same time period, the user base and hence the number of HDFS installations continuously increased. While we were not able to obtain exact numbers on the count of HDFS installations, we used as an alternative measure the level of interest in HDFS, as observed by the number of times HDFS appeared in Google searches (cyan line) [21]. The trends we observe likely indicate that for large-scale distributed systems, many bugs do not become exposed until the software is deployed at large scale across many users [22]. Finally, our observations might also be a manifestation of the "release early, release often" philosophy employed by many open-source developers and definitely by the HDFS community, with an average time between releases of less than 62 days, where developers rely on early customer feedback and bugs being exposed in the field, rather than lengthy testing before release.

B. Types of issues

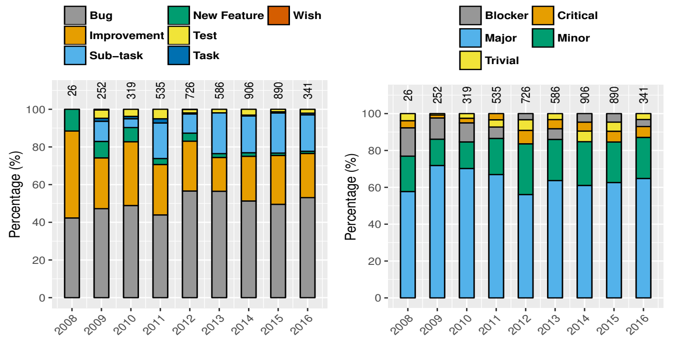
As described in Section II-C, JIRA [16] provides a *default* classification scheme, which categorizes reports based on their *type* and their *priority*. Initially, we present the classification results based on the specified default *type*. In this paper, we consider the most significant types that constitute the 93.67% of all *closed* and *resolved* reports. We chose to exclude some *types* from our study, because the provided description and developer comments were inconclusive; the missing categories are *New Feature*, *Test*, and *Wish*, comprising a total of 223 (6.33%) resolved reports. Figure 2a shows a timeline of the breakdown of all reports in the JIRA database by their *type*.

We observe that consistently over time, *bug*-related reports constitute the most common type of reports, making up 57% of all reports, and that their relative share of all reports keeps increasing over time. They involve issues related to the core implementation of HDFS, such as algorithmic errors and wrong configurations. This observation differs from the results of Antoniol *et al.* [23], which suggest that open-source systems may contain a larger fraction of non-bug issues.

The second largest category is reports related to *Improvements*. They make up (25.90%) of all reports, although their relative share has been slightly decreasing over time. They involve issues related to the system's performance and availability, along with implementation of new features.

Finally, the remaining categories comprise 22.61% of all reports. Depending on the description of the reported issue,

[2]We use the SLOCCount program [20] to count the lines of code and consider only Java source files in our statistics.



(a) *Per type.* (b) *Per priority.*

Fig. 2: *Breakdown results based on the default type (2a) and priority (2b).*

Task and *Sub-task* reports are eventually considered either as *Bug* or *Improvement* reports under our classification scheme.

C. Priorities of issues

Figure 2b shows a timeline of the breakdown of all reports in the JIRA database by their *priority*. Consistently over time, the most frequent category is *Major* issues, comprising 61.64% of all reports. These reports indicate significant loss of functionality and involve issues related to unexpected runtime failures, job failures, etc. The second most common category is *Minor* issues (21.33%), which involves minor loss of functionality and problems that can be easily resolved. We observe that their portion is non-negligible and therefore, we do not exclude them from our study (compared to Gunawi *et al.* [24]).

The most worrisome issues are those classified as *Blocker* (6.91%) and *Critical* (5.85%). *Blocker* reports must be resolved immediately, otherwise the development and testing procedures cannot make progress. *Critical* issues involve problems related to system crashes, loss of data, and memory leaks, which can have a significant impact on the system's reliability. We observe that the frequency of *Critical* issues increases over time. Finally, *Trivial* (4.27%) issues are the easiest to fix and their frequency remains relatively stable over the years.

D. Stable releases versus alpha/beta releases

HDFS follows a "release early, release often" strategy, with a significant number of alpha and beta releases, besides stable releases. We look at how effective this strategy is in detecting issues early, before they can make their way into a stable release. Towards this end, we consider what fraction of issues is being reported for *alpha* and *beta* releases versus stable releases. This percentage is reported for all classification *categories* described throughout the paper in Tables I and V. We observe that more than a third (33.71%) of all reported issues were reported for an alpha/beta release. Depending on the *type* of the issue, this rate can be significantly higher, such as 80% and 39% for issues related to maintenance and error codes respectively, or lower, as in the case for memory issues related to *null* pointers, or issues related to improper thread synchronization (17% and 19%, respectively). On the positive side, issues that are classified as *Critical* have a good (34%) chance of being detected during an alpha/beta release (as do *Minor* and *Trivial* issues), while unfortunately *Blocker* issues are less likely to be detected during alpha/beta releases (18%).

IV. Complexity of Issues

In this section, we are interested in the complexity of issues, which we study from three different angles, namely the size of an issue's scope, the amount of time it takes to detect an issue, and the time required to resolve it.

A. *Scope of issues*

One measure of an issue's complexity is how large its scope is. More precisely, we look at the size of the patch that is required to fix an issue, and how many files are affected by that patch.

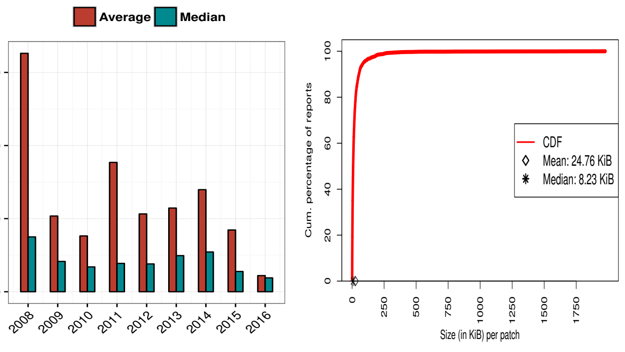


Fig. 3: *Distribution of the average size per patch file (in KB).*

Figure 3a shows the average and the median size (in kilobytes) of patches over time. We find that the median size of a patch is 8.2 KB and, despite the continuous increase in the overall size of the code base, the size of patches remains stable over the years, after a small decrease in value after the first year of development. The mean (25 KB) is significantly larger and inspection of the corresponding full CDF (see Figure 3b) reveals a long tailed distribution with the largest 1% of patches comprising more than 256 KB.

We also study how many files inside the code repository each patch modifies (see Figure 4a). We make use of the *diffstat* program [25] to calculate the number of affected files by a single patch. Again, we find surprising stability over time (past 2008), with a median of 3 and an average of 6.7 files affected by a patch, along with a long tail in its distribution (see Figure 4b); the largest 1% of patches affects more than 15 files.

Furthermore, we also explore how many lines were inserted and deleted by a patch (Figures 5a and 6a). The results remain stable after 2008, with a median of 72 and an average of 274.37 inserted lines, as well as a median of 13 and an average of 118.70 deleted lines by a patch. The corresponding distributions (Figures 5b and 6b) indicate that the largest 10% of patches insert and delete more than 540 and 190 lines of

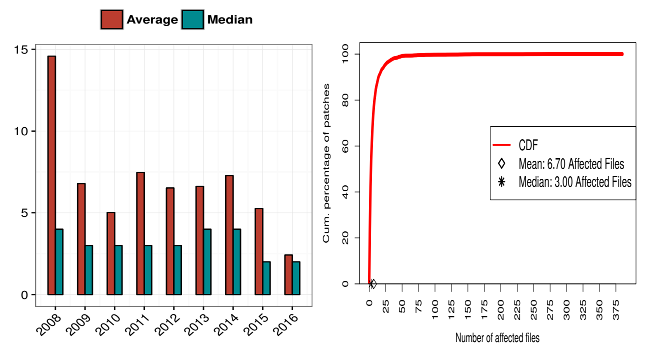


Fig. 4: *Distribution of the average number of files affected by a patch file.*

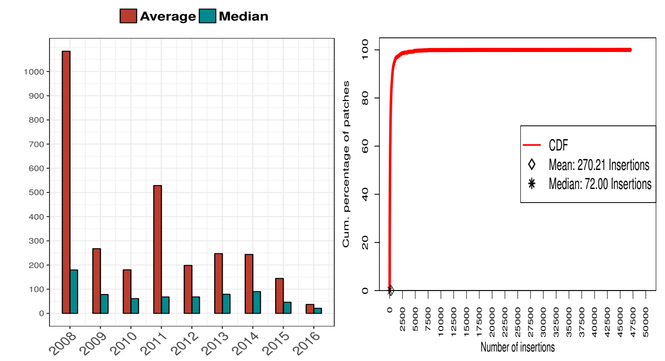


Fig. 5: *Distribution of the average number of insertions by a patch file.*

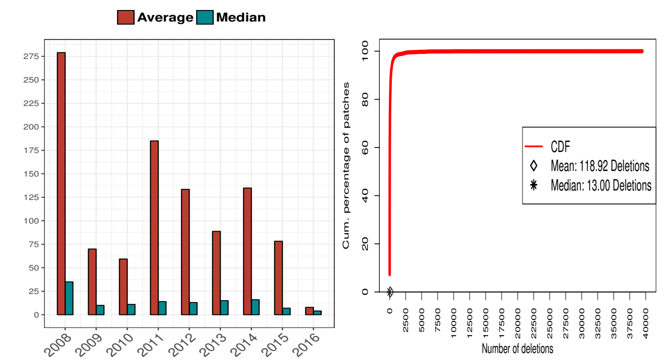


Fig. 6: *Distribution of the average number of deletions by a patch file.*

code, with a maximum value of $47,045$ and $39,479$ lines respectively.

Finally, in Figure 7a, we report the number of patches included in a single report. A report contains multiple patch files when the corresponding issue affects more than one

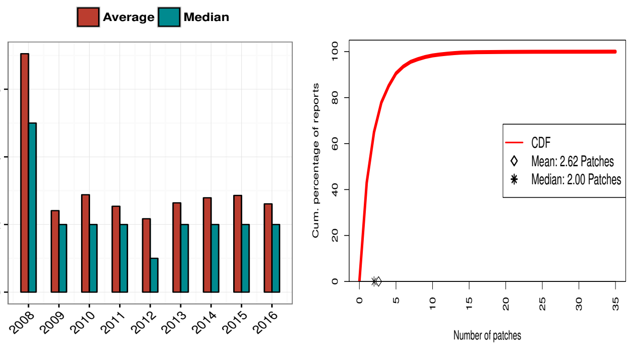


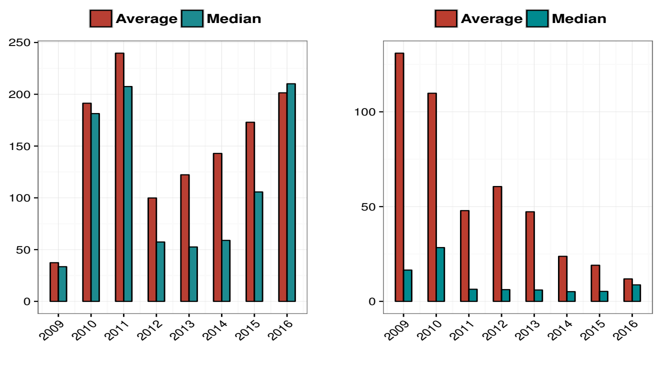
Fig. 7: *Distribution of the average number of patch files per report.*

development or release branches. We observe a median of 2 and a mean of 2.62 patches per report. The largest 10% of reports involve more than 5 patches, with a maximum value of 35 patch files in a single report. The corresponding report (*HDFS-347*) took developers many years to implement and contains multiple different patch versions.

In summary, our results indicate vast differences in the complexity of issues, some of which are easier to tackle, while the most complex ones affect large parts of the code base. Also, the median size of patches seems to be higher in the first year of development and then, drop.

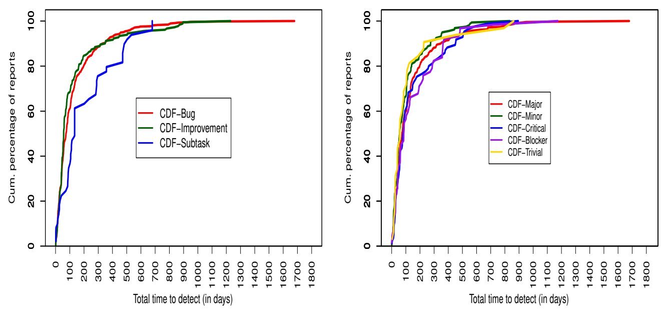
B. Time to detect issues

An interesting question is for how long an issue remains latent, i.e., it goes unnoticed until it is finally detected. To answer this question, we look at the creation timestamp of an issue in the JIRA archive and then, we look at the different released versions that are affected by this issue. We then take the timestamp of the release date of the oldest version affected by the issue and look at the difference between it and the time the issue was reported.



(a) Detection time. (b) Time to Resolve.

Fig. 8: The figure shows the time (in days) to detect an issue, i.e., the time between code being released and an issue being reported (Figure 8a) and the time to resolve an issue (Figure 8b).



(a) (b)

Fig. 9: *The time (in days) to detect an issue, i.e., the time between code being released and an issue being reported per type (Figure 9a) and priority (Figure 9b).*

As we are interested in the system's evolution over time, Figure 8a shows the average and median time to detection over time[3]. We find that while on average an issue is detected within 136.74 days (median 71.8 days), the time to detect an issue has been growing continuously over the years. We also broke down detection time by *type* and *priority* (Figures 9a and 9b respectively). We find that while detection time is similar for *Bug* and *Improvement* issues, there is a significant discrepancy in the detection time depending on the *priority*; *Critical* and *Blocker* spend 30-50% longer time in the field before they are detected, compared to *Trivial* and *Minor* issues. Finally, 20% of *Critical* and *Blocker* issues are detected more than 1 year after the code they affect was released.

C. Time to resolve issues

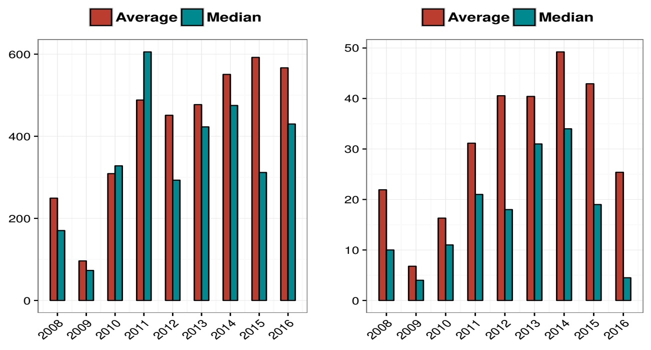
The next question we ask is how long it takes to resolve an issue once it has been detected. We find that half of all reports are resolved within less than 6.91 days, while the average is at 52.3 days significantly higher than that. The discrepancy between median and mean indicates a long-tailed distribution, where most reports can be resolved very quickly, while some take a very long time. We also find that the difference in time to resolve a *Bug* versus an *Improvement* issue is small, while there is a significant difference depending on the *priority* of an issue; the median time to resolve a *Trivial* or *Minor* issue is only 1.5 and 4.8 days respectively, while for resolving *Critical*, *Major* and *Blocker* issues, it ranges from 7 to 9.6 days.

Interestingly, we observe that the time to resolve an issue has decreased significantly over the years, as shown in Figure 8b, despite the fact that the complexity of issues has not decreased (recall Section IV-A). One possible explanation is that the number of developers involved in fixing issues has increased over time (see purple line in Figure 1)[4]. This explanation is supported by our observation that also the time it takes until someone starts working on an issue, once it has been reported, continuously decreases over time,

[3]We omit years 2009-2011 as for these years, only very few reports have information on which versions were affected by each report.

[4]We omit year 2008 because it involves only a few initial releases.

from around 30 days in 2010 to less than 5 days in 2015. However, we also observe that the growth in size of the HDFS code base outstrips the growth in number of active developers. We therefore examine whether the reduction in time to resolve an issue is due to a combination of a more engaged group of developers, as well as developers becoming more experienced and hence, faster at resolving issues. We measure the experience of a developer by calculating the number of previously fixed bugs, along with their experience in days, i.e., the number of days elapsed from the first fixed report to the current report's resolve date. The corresponding figures (see Figures 10a and 10b) indicate that up to 2014, developers become more and more experienced, reaching a maximum median value of 32 fixed reports and 598.27 days of experience. After that, the corresponding values decrease by almost a half. Finally, in Tables I and V, we include detailed statistics regarding the experience of the assigned developers per classification *category*.



(a) (b)

Fig. 10: *The experience (in days) of HDFS developers (Figure 10a) and the total number of fixed reports (Figure 10b) per year.*

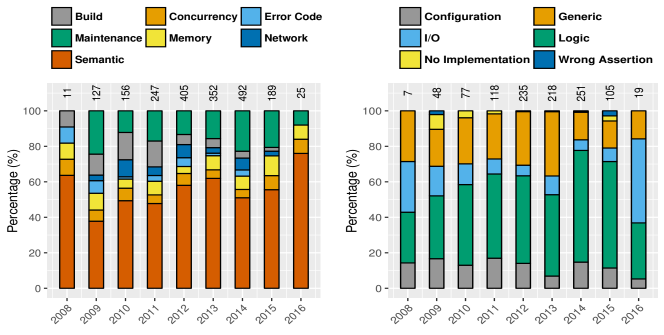
V. A closer look at Bug Reports

In this section, we initially present our mechanism for classifying *Bug* reports into different *sub-categories*, and then we describe in detail the three most important *sub-categories*.

A. Breaking down Bug Reports

As *Bug* reports constitute the most dominant *category* in HDFS, we ask what is the most common source of errors among them. Figure 11a and Table I present the breakdown of all bug reports into the *sub-categories* of our two-level hierarchy. We manually inspected and classified each individual report, by taking into consideration the corresponding description, comments, and provided patch files.

We find that *Semantic* bugs are responsible for more than half of all bug reports (53.79%) and that their relative frequency is slowly increasing over time. The bugs of this *category* are directly related to the core functionality of the file system. The procedure to detect and repair these bugs requires low-level knowledge about the system itself, since *Semantic*



(a) *Bug reports.* (b) *Semantic reports.*

Fig. 11: *Breakdown of* Bug *reports based on their classification type (11a) and its largest sub-category, Semantic reports (11b).*

bugs involve algorithmic errors, incorrect assumptions about the system's state, invalid configurations, etc. As a result, it comes as no surprise that this category contains the highest number of assigned developers (192).

The next most common *category* is Maintenance (17.66%). This category involves API changes, elimination of warnings, updates in test methods, documentation issues, along with other modifications that do not necessarily affect the core functionality of the file system itself.

The remaining reports are split somewhat equally into *Memory*, *Concurrency*, *Network*, *Build*, and *Error Code*. *Memory* reports (7.19%) contain issues related to memory allocation and handling. *Concurrency* reports (5.79%) include common issues related to multi threaded programming and is the category of bugs that takes the longest time to detect and to resolve. *Network* reports (5.49%) involve errors related to data transmission and network authentication. They are generally assigned to experienced developers that have a median value of 476.71 days of experience and 24.5 previously fixed bugs, possibly due to their complex nature and their difficulty to reproduce.

Build reports (6.99%) contain errors related to the system's setup, deployment and execution, such as errors in execution scripts and XML files. *Build* bugs seem to be a relatively benign category in that, it is the one that contains issues with the shortest time to detect and resolve (possibly due to its importance to deployment), a high fraction of its issues are detected during alpha/beta releases rather than stable releases, and its rate is decreasing over time. Finally, *Error Code* reports (3.09%) aim to improve the system's error propagation by fixing the error codes returned during various function calls and adding missing return statements.

B. Semantic Bugs

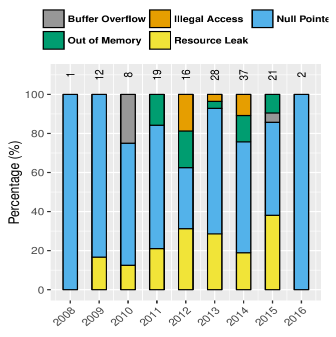
This subsection takes a closer look at *Semantic* bugs, as they constitute more than half of all HDFS bugs. The complete list of all distinct classification *sub-categories* related to *Semantic* reports is shown in Table II, while Figure 11b shows a breakdown over time.

Name	**Description**	**Percentage**	**Alpha/Beta**	**Resolve Time**	**Detection Time**	**Assigned Developers (Total - Experience Days - Fixed Bugs)**		
Build	Project build bugs, such as missing scripts and compilation errors.	6.99%	38.71%	$43 - 4.12$	$87.35 - 45.62$	66	$311.99 - 159.83$	$22.28 - 6.5$
Concurrency	Bugs related to concurrency, such as atomicity, synchronization, etc.	5.79%	25.86%	$77.57 - 12.42$	$179.79 - 80.05$	53	$520.15 - 430.03$	$36.89 - 15.5$
Error Code	Wrong or missing error codes.	3.09%	24.86%	$66.88 - 9.21$	$146.63 - 74.23$	33	$489.53 - 234.28$	$46.32 - 21$
Maintenance	Maintenance of documentation and test code (files, APIs, etc.).	17.66%	19.9%	$49.36 - 5.17$	$108.82 - 58.76$	106	$419.35 - 247.93$	$32.08 - 14$
Memory	Issues regarding memory allocation, handling and release.	7.19%	30.90%	$50.66 - 6.06$	$150.44 - 62.93$	65	$510.16 - 383.99$	$37.63 - 18$
Network	Bugs related to network protocols; issues related to authentication.	5.49%	24.29%	$33.67 - 6.84$	$112.9 - 75.13$	46	$568.42 - 476.71$	$38.53 - 24.5$
Semantic	Bugs in the core implementation of the file system.	53.79%	32.19%	$53.03 - 6.72$	$137.78 - 81.97$	192	$502.06 - 400.57$	$37.61 - 20$

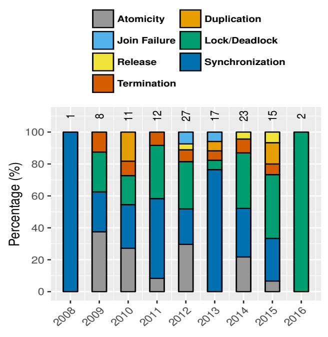
TABLE I: *The classification categories of Bug reports, along with the corresponding resolve and detection times (in days). For those columns that contain two values, these values represent average and median respectively.*

Name	**Description**	**Percentage**	**Resolve Time (Avg - Median)**
Configuration	Wrong, missing, or inconsistent configurations.	12.71%	$61.78 - 6.12$
Generic	Fixing typos, minor mistakes and minor changes in code.	25.05%	$47.72 - 2.49$
I/O	Issues related to read, write, open, close calls for files.	9.09%	$63.52 - 12.39$
Logic	Wrong implementation and general errors in algorithms.	51.30%	$51.44 - 8.83$
No Implementation	Lack of implementation.	1.39%	$46.01 - 3.08$
Wrong Assertion	Errors thrown by wrong assertions.	0.46%	$92.18 - 1.35$

TABLE II: *The classification sub-categories of Semantic reports, along with the corresponding time to resolve (in days).*



(a) *Memory reports.*



(b) *Concurrency reports.*

Fig. 12: *Breakdown of* Memory *reports (12a) and* Concurrency *reports (12b).*

We observe that *Logic* is the most common *sub-category*, occupying over half of the *Semantic* bugs (51.30%) and that its frequency is growing over time. This *sub-category* includes bugs related to wrong assumptions about the file system in general, wrong implementation of algorithms, and bad design choices. An example involves the wrong update of a file's modification time when a particular class is reloaded (*HDFS-1138*). In this *sub-category*, the developer's experience is critical while resolving the corresponding bugs.

Furthermore, *Generic* errors comprise a large percentage of *Semantic* bugs (25.05%), followed by *Configuration* bugs (12.71%). In their majority, *Generic* reports comprise of minor errors that can be resolved without requiring much effort and domain knowledge. They are usually caused by a programmer's negligence, such as the wrong invocation of a method due to similar method names. *Configuration* bugs involve issues related to wrong usage or wrong initialization of configuration parameters. Such parameters typically reside in the *"configuration"* package of the HDFS source code repository.

Finally, 9.09% of *Semantic* bugs are related to read and write file operations and are classified as *I/O* bugs. These also include issues related to open and close operations on files. The bugs of this *sub-category* include I/O malfunctions of the file system itself, as well as, problematic I/O operations and configurations for other components and functions, such as read configurations.

In summary, *Logic* reports, which constitute the most dominant *sub-category* among *Semantic* bugs and require experienced developers to resolve, grow in frequency over time. On the other hand, *Generic* errors can be resolved more easily, and therefore, experience a decrease in frequency over time.

Name	**Description**	**Percentage**	**Resolve Time (Avg - Median)**
Buffer Overflow	Exceed a buffer's size.	2.08%	$11.4 - 9$
Illegal Access	Array index out of bounds.	5.56%	$12.11 - 5.39$
Null Pointer	Dereference of *null* pointers.	58.33%	$63.47 - 7.14$
Out of Memory	Memory exhaustion.	9.72%	$15.22 - 3.27$
Resource Leak	Memory release/GC errors.	24.31%	$46.27 - 6.08$

TABLE III: *The classification sub-categories of Memory reports, along with the corresponding time to resolve (in days).*

C. Memory Bugs

This subsection takes a closer look at *Memory* bugs, as they are the second largest bug category and the one with the second longest time to detect. *Memory* bugs occur due to a number of reasons. Buffer overflows are caused by improper copy operations that exceed the maximum size of a buffer. The dereference of a *null* pointer can result in a system crash. Our categorization is highly based on the consequences imposed by such bugs and therefore, the classification procedure is straightforward. The complete list of all distinct classification *sub-categories* related to *Memory* reports is shown in Table III, while Figure 12a shows a breakdown over time.

Name	Description	Percentage	Resolve Time (Avg - Median)
Atomicity	The atomicity property for thread access is violated.	18.10%	130.63 – 23.88
Duplication	Remove redundant processes.	4.30%	10.58 – 10.15
Join Failure	Not joined threads; incorrectly terminated processes.	2.59%	4.3 – 1.86
Lock/Deadlock	Wrong usage of locks.	28.45%	34.26 – 10.22
Release	Obtained locks are not released properly.	2.59%	5.92 – 2.03
Synchronization	Improper synchronization.	36.21%	90.62 – 11.96
Termination	Incorrectly killed processes, or processes that should be killed but are not.	7.76%	137.15 – 21.17

TABLE IV: *The classification sub-categories of Concurrency reports, along with the corresponding time to resolve (in days).*

We observe that *Null Pointer* bugs are the most common among all *Memory* bugs (58.33%). A missing initialization, or a wrong assumption about the return value of a function can often lead to a *null* pointer exception being thrown by the Java Virtual Machine (JVM).

Additionally, there is a fair number of *Resource Leak* errors (24.31%) for which we observe that their frequency is increasing over time. Similar to traditional file systems, resource leaks in HDFS often refer to unreleased allocated blocks that are no longer used by the file system itself. Furthermore, improper handling of memory allocations can lead to memory leaks as well. For example, a process may neglect to deallocate unused portions of memory or close unused sockets.

The remaining three sub-categories are relatively rare. Java itself reports *Out of Memory* bugs (9.72%) by throwing the corresponding exception. In such cases, the running process has exhausted the available portion of memory and has eventually terminated. Furthermore, *Illegal Access* bugs (5.56%) contain mostly exceptions thrown to indicate that an index is out of range. The index may be related to an array, a Java string, etc. Finally, *Buffer Overflow* bugs (2.08%) are caused by improper copy operations that exceed the maximum size of a buffer.

In summary, we conclude that *Null Pointer* errors are the dominant type of memory related bugs, emphasizing the importance of code analysis tools (either static or dynamic) that detect such kind of errors. Moreover, the frequency of *Resource Leak* errors continuously increases over time, indicating the difficulty of proper memory management in a continuously growing code base.

D. Concurrency Bugs

This subsection takes a closer look at *Concurrency* bugs, as they are the third-largest category of *Bug* reports and the one that takes the longest time to detect and resolve. We partition all *Concurrency* bugs into several *sub-categories* based on the provided description, comments and patch files for each individual report. The complete list of all distinct classification *sub-categories* related to *Concurrency* reports is shown in Table IV, while Figure 12b shows a breakdown over time.

The two most dominant *sub-categories* among the HDFS *Concurrency* bugs are *Synchronization* and *Lock/Deadlock*, with 36.21% and 28.45% of the total *Concurrency* reports respectively. *Synchronization* reports include join failures, inappropriate heartbeat timeouts, and deprecated ordering of multiple accesses. *Lock/Deadlock* reports involve examples where a process uses the wrong lock, or a process fails/misses to release an acquired lock, or a deadlock is caused by inconsistent ordering of locks across threads.

The *Atomicity* category (18.10%) captures violation of thread safety, which can lead to incorrect return values or possible race conditions. These bugs are among the ones that take on average the longest time to resolve (130 days on average) among all *Concurrency* bugs. In general, *Synchronization*, *Lock/Deadlock* and *Atomicity* bugs are hard to detect. For this reason, developers tend to implement special unit tests to simulate a failure situation and to verify the system's correctness under extreme scenarios.

Termination (7.76%) and *Duplication* (4.30%) reports include management issues at a thread granularity. *Termination* reports include examples where active threads are not terminated properly, while *Duplication* reports include examples where duplicate threads are created to perform the same task. The percentage of *Termination* errors remains stable over time, indicating the need for better testing and cleanup procedures.

Finally, there are a few *Join Failure* (2.59%) and *Release* (2.59%) reports, where terminating threads are not joined properly and running threads do not release the allocated resources respectively.

In summary, *Synchronization* and *Lock/Deadlock* errors dominate *Concurrency* bugs. However, other types of *Concurrency* bugs, in particular *Atomicity*, *Termination* and *Duplication* bugs, also consistently appear over the years and all contribute to the long time of detection and resolution of *Concurrency* bugs.

VI. A CLOSER LOOK AT IMPROVEMENT REPORTS

In this section, we focus on *Improvement* reports. We begin by classifying them into different *sub-categories*, and then we describe in detail the four most important *sub-categories*.

A. Breaking down Improvement Reports

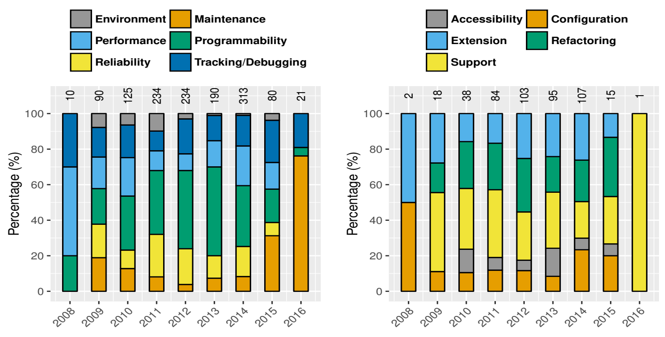
Improvement reports can be associated with different aspects of HDFS. Figure 13a and Table V show a breakdown of such reports into the different categories we have identified.

We observe that the most dominant category is *Programmability* (35.70%), which includes reports that aim to implement and provide new features, extract more information from the system, re-structure the source code and finally, provide support for new mechanisms. More than a third of *Programmability* issues can be identified during alpha/beta releases, rather than later for stable releases. This category also contains the highest number and the most experienced developers among all *Improvement* reports.

The next most common targets of improvement are *Reliability*, *Tracking/Debugging* and *Performance*, each of which accounts for around 16% of all *Improvement* reports. *Reliability* reports aim to enhance the system's security and robustness

Name	**Description**	**Percentage**	**Alpha/Beta**	**Resolve Time**	**Detection Time**	**Assigned Developers (Total - Experience Days - Fixed Bugs)**		
Environment	Modifications in XML, build and shell files.	4.09%	15.09%	47.38 − 14.99	132.43−120.38	28	304.38 − 95.95	21.09 − 6
Maintenance	Both code and documentation.	10.95%	23.94%	71.09 − 7.65	157.54−106.24	58	450.92−214.08	30.96 − 16
Performance	Optimized algorithms, I/O strategies and scheduling.	15.88%	19.90%	94.94 − 14.99	215.15 − 69.01	68	501.73−405.08	41.82 − 23
Programmability	New mechanisms, API development, code refactoring.	35.70%	34.77%	64.35 − 7.51	110.39 − 59.35	92	577.29−551.89	48.35 − 34
Reliability	Improvements in the file system's robustness and availability.	16.65%	24.54%	28.91 − 3.88	113.75 − 56.87	54	534.96−461.59	48.54 − 35
Tracking/Debugging	Enhancements in logs and the web interface, extended error reports.	16.73%	22.12%	88.65 − 12.72	169.82 − 55.09	81	399.19−229.92	29.59 − 16

TABLE V: *The classification categories of Improvement reports, along with the corresponding resolve and detection times (in days). For those columns that contain two values, these values represent average and median respectively.*



(a) *Improvement reports.* (b) *Programmability reports.*

Fig. 13: *Breakdown of Improvement reports based on their classification type (13a) and its largest sub-category, Programmability reports (13b).*

properties by i) removing unused code, ii) implementing new mechanisms to detect failures among the system's components, and iii) improving the availability of the system in extreme scenarios of execution. The developers of this category are more experienced, having a median value of 35 previously solved reports.

The *Tracking/Debugging* category (16.73%) contains reports related to the system's logging infrastructure. They aim to clarify the displayed information by specifying how descriptive and analytical the logs should be. It is important to tune the frequency of log operations, since logs can become quite big and can significantly affect the system's performance by issuing multiple I/O requests. Moreover, this category makes up a significant share of all *Improvement* reports for every single year in the HDFS lifecycle and it is among the two *sub-categories* with the largest time to detect and to resolve.

Reports in the *Performance* category (15.88%) aim to increase the performance of HDFS by modifying the current core implementation, by removing unnecessary operations, by performing smarter scheduling, and by adopting a parallel model of execution when possible. Issues in this category are the ones that on average take the longest to detect (215 days) and the longest to resolve (95 days) among all *Improvement* reports.

Maintenance reports (10.95%) involve minor changes in the system's source code repository. They are related to enhanced documentation, renaming of variables and methods for improved readability, and more thorough testing. Finally, *Environment* reports (4.09%) tackle issues related to build files, execution scripts, and XML configuration files, requiring only a few developers without significant experience.

In summary, *Programmability* reports constitute the most dominant *Improvement* category, occupying more than a quarter of such reports, while improvements related to *Tracking/Debugging* and *Performance* are the ones that on average take the longest time to identify and to resolve, potentially because these issues mostly manifest and become most challenging once installations reach a large-scale.

B. *Improvements of Programmability*

This subsection takes a closer look at *Programmability* reports, whose main goals are first, to enhance the development procedure and second, to provide the developers and users of the system with additional mechanisms and features. The complete list of all distinct classification *sub-categories* related to *Programmability* reports is shown in Table VI, while Figure 13b shows a breakdown over time.

Many reports involve the implementation of a new feature; however, this may require the development of additional minor mechanisms in order for HDFS to fully support the new feature. Furthermore, this *category* aims to provide new APIs and methods to extract information from the underlying infrastructure, such as additional metrics, etc. The additional information can be used for debugging purposes and also, for more accurate analysis and evaluation of the system. We carefully study all *Programmability* reports and further classify them into specific *sub-categories*.

Name	**Description**	**Percentage**	**Resolve Time (Avg - Median)**
Accessibility	New APIs to extract additional information.	8.64%	34.21 − 11.16
Configuration	Make hard-coded parameters configurable.	14.04%	106.33 − 5.88
Extension	Mechanisms and methods to improve usability.	22.68%	76.7 − 18.83
Refactoring	Refactoring of code for better structure and organization.	24.84%	29.01 − 4.12
Support	Provide support for other mechanisms, consistency with existing semantics.	29.81%	73.38 − 10.1

TABLE VI: *The classification sub-categories of Programmability reports, along with the corresponding time to resolve (in days).*

We observe that the most *Programmability* improvement reports are related to *Refactoring* (24.84%), *Extension* (22.68%) and *Support* (29.81%). *Extension* reports introduce new mechanisms and methods to improve usability. *Refactoring* reports aim to restructure the code to improve the general organization, while *Support* reports provide support for other mechanisms and improve the system's consistency with respect to existing semantics.

To a smaller degree, *Programmability* reports also aim to improve support for *Configuration* (14.04%), for instance, by making hard coded parameters configurable. Finally, *Accessibility* features aim to provide an API to extract additional information from the underlying infrastructure, such as metrics and statistics.

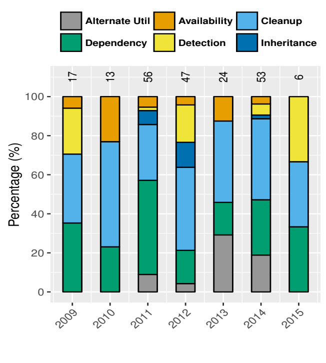
In conclusion, we observe that a significant portion of the development procedure is dedicated to i) the system's expansion with new mechanisms and ii) the refactoring of operations to improve the code structure in general.

C. Improvements of Reliability

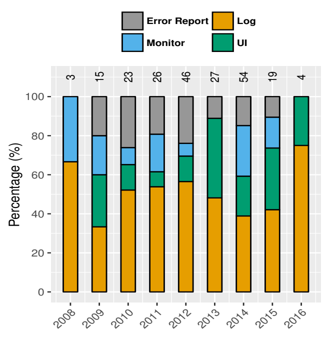
Reliability reports aim to enhance the system's availability, security and robustness properties. The complete list of all distinct classification *sub-categories* related to *Reliability* reports is shown in Table VII, while Figure 14a shows a breakdown over time.

Name	**Description**	**Percentage**	**Resolve Time (Avg - Median)**
Alternate Utilities	Invoke different methods that provide the same utility.	11.11%	45.58 – 2.42
Availability	Improve system availability.	6.48%	57.74 – 26.11
Cleanup	Removal of unused or unnecessary functionality.	38.43%	22.43 – 1.62
Dependencies	Alter the dependencies among packages.	30.09%	19.78 – 5.2
Detection	Mechanisms to detect failures.	8.80%	28.89 – 21.01
Inheritance	Issues related to public, protected, private identifiers.	5.09%	58.71 – 2.16

TABLE VII: *The classification sub-categories of Reliability reports, along with the corresponding time to resolve (in days).*



(a) *Reliability reports.*



(b) *Tracking/Debugging reports.*

Fig. 14: *Breakdown of* Reliability *reports (14a) and* Tracking/Debugging *reports (14b).*

We observe that, by a wide margin, the largest categories are *Cleanup* (38.43%) reports, which remove unnecessary methods and code segments from the HDFS code repository or move files to different packages in order to provide a safer and more robust environment, and *Dependencies* (30.09%), which restructure the code repository to avoid unnecessary dependencies among packages.

The smaller categories are *Alternate Utilities* reports (11.11%), which change the code to invoke different methods that provide the same utility, *Detection* reports, which implement new mechanisms for detecting failures, *Availability* reports (6.48%), which aim to improve system availability, and *Inheritance* reports, which deal with issues related to public, protected and private identifiers.

We notice that the frequency of reliability reports, and their breakdown, do not significantly change over time, indicating that continuous efforts are necessary in order to keep a large code base reliable.

D. Improvements of Tracking/Debugging

This section takes a closer look at *Tracking/Debugging* reports, which mostly involve reports related to the system's logging infrastructure. The complete list of all distinct classification *sub-categories* related to these reports is shown in Table VIII, while Figure 14b shows a breakdown over time.

Name	**Description**	**Percentage**	**Resolve Time (Avg - Median)**
Error Reporting	Modifications in error codes and exception messages.	17.51%	69.27 – 9.76
Logging	Issues related to logs. Display additional information.	47.93%	79.14 – 6.44
Monitoring	Issues related to real-time monitoring, consoles, and standard output streams.	14.29%	70.72 – 18.02
Web Interface	Issues related to the web interface. Display additional information.	20.28%	140.48 – 18.59

TABLE VIII: *The classification sub-categories of Tracking/Debugging reports, along with the corresponding time to resolve (in days).*

Nearly half of all reports in this *category* are concerned with the *Logging* (47.93%) of operations and messages, in order to assist programmers during the debugging procedure and users during system operation. The remaining three *sub-categories* contain approximately the same number of reports, namely reports related to the Web Interface (20.28%) and its display of additional information, reports improving *Error Reporting* (17.51%) through additions and modifications related to error codes and exception messages, and *Monitoring* (14.29%) reports, which aim to provide administrators with real-time monitoring of the system.

In summary, we conclude that the need to display additional information in the system's logs and in the system's web interface is constantly increasing over the years. On the other hand though, the number of issues related to error codes and exception messages has been decreasing over the years, indicating that the developers of HDFS are trying hard to provide correct and detailed information to the users of the system.

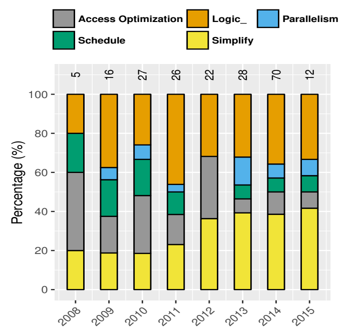


Fig. 15: *Breakdown of* Performance *reports.*

E. Improvements in Performance

This section focuses on *Performance* reports, which involve modifications in the core functionality of HDFS and in the existing implementations of algorithms. As we observed earlier, the issues of this *sub-category* tend to be among the ones that take the longest to identify and to resolve. The complete list of all distinct classification *sub-categories* related to these reports is shown in Table IX, while Figure 15 shows a breakdown over time.

Name	**Description**	**Percentage**	**Resolve Time (Avg - Median)**
Access Optimization	Smart data and I/O access strategies.	16.99%	169.92 − 42.91
Logic	Modifications in the existing implementations.	34.47%	81.77 − 23.52
Parallelism	Parallel execution for performance and scalability.	6.80%	104.11 − 52.54
Scheduling	Thread synchronization, starvation, and priority issues.	9.71%	151.7 − 18.65
Simplification	Remove complicated logic.	32.04%	50.4 − 4.04

TABLE IX: *The classification sub-categories of Performance reports, along with the corresponding time to resolve (in days).*

About a third of the reports target *Simplification* (32.04%) of existing code, often by removing unnecessary operations and method invocations that affect the performance of the system, and roughly another third, *Logic* (34.47%) reports strive to increase performance by improving existing implementations of algorithms in the core functionality of HDFS. The remaining reports in the *Performance category* optimize data access and I/O strategies (*Access Optimization*, 16.99%), try to exploit the parallel execution of threads for better performance (*Parallelism*, 6.80%), along with enhanced tuning of thread priorities and synchronization (*Scheduling*, 9.71%).

VII. Related Work

While there has been a long-standing interest in the software engineering community to mine code repositories, we are aware of only two prior studies involving the Hadoop File System. Gunawi *et al.* [24] perform a comprehensive study of development issues of six well-known cloud systems, including HDFS. Their analysis is focused on 3655 "vital" issues across the six systems (with HDFS being the only file system in the study). They present a wide range of bugs unique to distributed cloud systems, such as scalability, topology, and killer bugs. Huang *et al.* [26] present a study of issues in the HDFS and in the MapReduce platforms of the Hadoop framework. They examine a number of different issues, with a focus on correlations, consequences, impacts, and reactions. Our work is complementary since we also focus on HDFS, but with a different focus. We provide a more fine-grained classification scheme for issue reports and also, study the complete history of the system including **all** released versions, reports and patch files. Moreover, we provide a detailed analysis of characteristics that have not been previously presented in detail for distributed file systems, such as the scope and complexity of issues in terms of the size of the patch that fixes it and the number of files it affects, the time it takes before an issue is exposed, and issues detected during alpha/beta releases versus stable releases, among others.

In [27], Lu *et al.* present their quantitative study of 5079 patches across six Linux file systems. They focus on open-source local file systems and they examine every file system patch in the Linux 2.6 series. Their results indicate that nearly half of the patches are maintenance patches, while the next dominant category is bugs. Our study is complementary since we also analyze patches, but instead, we focus on HDFS and examine its evolution over time.

In [28], Yuan *et al.* present an in-depth analysis of 198 user-reported failures in some widely used distributed systems. They focus on understanding the failure manifestation sequences and how errors evolve into component failures and service-wide catastrophic failures. Unlike our work, which involves an open-source distributed system, [28] does not cover HDFS or any other file system at all. Moreover, instead of considering only bug reports, we study the entire history and evolution of HDFS, examining all reports and patches in detail.

There are also studies that have been performed for various other systems, other than file systems. Chou *et al.* present a study of operating system errors found by automatic static analysis applied to the Linux and OpenBSD kernels [29]. Lu *et al.* perform a comprehensive study of 105 randomly selected real world concurrency bugs from 4 server and client open-source applications [30]. Yin *et al.* present their real-world study that involves 546 misconfigurations from a commercial storage system and 4 open-source systems [31].

Finally, large software systems and their fault characteristics have been extensively studied in contexts, other than HDFS [11], [12], [29], [30], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51]. These studies provide important results and insights for tool designers and software system developers. Closest to our work is the one by Lin *et al.* [11], [12], which presents a study on bug characteristics in three large open-source software projects, namely Mozilla, Apache, and Linux kernel. Unlike ours, their work is focused on bugs only and also differs in some of the findings; we find a significantly smaller fraction of bugs to be either semantic or memory bugs.

Moreover, Zaman *et al.* [52] also use Mozilla Firefox as their case study to discover how different types of bugs differ from each other in terms of time to resolve, number of developers, and number of affected source code files. In our study, we also take into account the number of affected source code files per patch, the number of developers per year, and the total time required to solve an issue. However, we focus on a distributed file system rather than a large application.

VIII. Conclusions

We performed a comprehensive study of 3302 reports and a total of 8280 patches across all released versions of the Hadoop Distributed File System (HDFS) over its first nine years. Our analysis includes those reports that are marked either as *closed* or *resolved* and contain *at least one* patch. The purpose of this study is to assist developers in improving the design of similar file systems and in implementing more solid and robust systems. Our results indicate that bug reports are the most dominant type and that they steadily increase over time, while the overall scope and complexity of reports and patches remain stable throughout HDFS' lifetime. We believe that our results and observations presented throughout the paper can provide insight into the evolution of distributed file systems and hope that they will spur further research in this area.

References

[1] "Apache Hadoop," https://hadoop.apache.org/.

[2] A. Inc., "Altior's AltraSTAR Hadoop Storage Accelerator and Optimizer Now Certified on CDH4 (Cloudera's Distribution Including Apache Hadoop Version 4)," http://www.prnewswire.com/news-releases/altiors-altrastar---hadoop-storage-accelerator-and-optimizer-now-certified-on-cdh4-clouderas-distribution-including-apache-hadoop-version-4-183906141.html, Dec. 2012, last Retrieved: May 23rd, 2016.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[4] "Hadoop MapReduce," https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: mazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[7] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[9] "Hadoop Distributed File System (HDFS) architecture guide," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium*. IEEE, 2010, pp. 1–10.

[11] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 25–33.

[12] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[13] M. A. Hartz, E. L. Walker, and D. Mahar, *Introduction to software reliability: a state of the art review*. The Center, 1996.

[14] "Hadoop MapReduce," https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[15] "Apache HBase," https://hbase.apache.org/.

[16] "Apache Software Foundation - JIRA issue tracker repository," https://issues.apache.org/jira/secure/Dashboard.jspa.

[17] "The Apache Software Foundation - Blogging in Action," https://blogs.apache.org/bigtop/entry/all_you_wanted_to_know.

[18] "Apache HDFS Change Log," https://issues.apache.org/jira/browse/HDFS?selectedTab=com.atlassian.jira.jira-projects-plugin:changelog-panel&allVersions=true.

[19] "Postgresql rdbms." http://www.postgresql.org/.

[20] D. A. Wheeler, "Sloccount," 2001.

[21] "Google Trends - hdfs," https://www.google.com/trends/explore?date=2008-09-01%202016-10-17&q=HDFS&hl=en-US.

[22] I. Herraiz, E. Shihab, T. H. Nguyen, and A. E. Hassan, "Impact of installation counts on perceived quality: A case study on debian," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 219–228.

[23] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[24] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[25] "The diffstat program," http://linux.die.net/man/1/diffstat.

[26] J. Huang, X. Zhang, and K. Schwan, "Understanding issue correlations: a case study of the Hadoop system," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 2–15.

[27] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of Linux file system evolution," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, p. 3, 2014.

[28] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 249–265.

[29] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 73–88.

[30] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM Sigplan Notices*, vol. 43, no. 3. ACM, 2008, pp. 329–339.

[31] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 159–172.

[32] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability: A study of field failures in operating systems," in *FTCS*, 1991, pp. 2–9.

[33] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. IEEE, 1992, pp. 475–484.

[34] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*. IEEE, 2000, pp. 97–106.

[35] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 485–494.

[36] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st*

International Conference on Software Engineering. IEEE Computer Society, 2009, pp. 298–308.

[37] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[38] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve," in *Proceedings of the 13th international conference on Software engineering*. IEEE Computer Society Press, 1991, pp. 246–255.

[39] A. Endres, "An analysis of errors and their causes in system programs," in *ACM Sigplan Notices*, vol. 10, no. 6. ACM, 1975, pp. 327–336.

[40] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[41] R. L. Glass, "Persistent software errors," *IEEE Transactions on Software Engineering*, no. 2, pp. 162–168, 1981.

[42] W. Gu, Z. Kalbarczyk, R. K. Iyer, Z.-Y. Yang *et al.*, "Characterization of linux kernel behavior under errors." in *DSN*, vol. 3, 2003, pp. 22–25.

[43] M. Kaâniche, K. Kanoun, M. Cukier, and M. B. Martini, "Software reliability analysis of three successive generations of a switching system," in *European Dependable Computing Conference*. Springer, 1994, pp. 471–490.

[44] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[45] K.-H. Moller and D. J. Paulish, "An empirical investigation of software fault distribution," in *Software Metrics Symposium, 1993. Proceedings., First International*. IEEE, 1993, pp. 82–90.

[46] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 55–64.

[47] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *Journal of Systems and Software*, vol. 4, no. 4, pp. 289–300, 1984.

[48] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[49] M. Pighin and A. Marzona, "An empirical analysis of fault persistence through software releases," in *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*. IEEE, 2003, pp. 206–212.

[50] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 465–475.

[51] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 26–36.

[52] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.