The Journal of Systems and Software 231 (2026) 112643

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

From description to prescription: Unraveling log severity adjustments in open-source software ☆

Eduardo Mendes [a,*], Marcelo Vasconcellos [a], Fabio Petrillo [b], Sylvain Hallé [a]

[a] Université du Québec a Chicoutimi, Saguenay, Canada
[b] École de Technologie Supérieure, Montréal, Canada

ARTICLE INFO

Dataset link: https://doi.org/10.6084/m9.figshare.26253776.v2

Keywords:
Log severity level
Log severity level adjustments
Misclassification
Log analysis
Logging
Software engineering

ABSTRACT

Context: Logs are vital to understanding a software system's behavior, often being the only evidence available to investigate failures.
Problem: Selecting a Log Severity Level (LSL) can be challenging for the following reasons: (i) the absence of knowledge about how logs are used in production, (ii) the lack of understanding of how critical an event is, and (iii) the lack of practical guidelines. This leads to frequent LSL adjustments during software development and evolution.
Objective: Our goal is to investigate the LSL adjustments between system releases and explore methods to improve LSL classification.
Methods: We analyzed the log statements from different releases of open-source systems, focusing on their LSL adjustments and examining the commit comments to understand the reasons for the adjustments.
Results: Our results show that most adjustments occur at the intersection of development and production environment logs. Furthermore, the main guiding factors for the adjustments are the experience and logging theory. Our contributions are (i) a description of trends and patterns in LSL adjustments and (ii) a set of 24 heuristics to guide the choice, review, and adjustments of LSL. We advise developers to adhere to the LSL purposes, routinely review LSL settings, and remain adaptable to their mutability.

1. Introduction

Logs are often the primary source of information for operators to understand and diagnose the behavior of a software system (El-Masri et al., 2020). In many cases, logs are the only available evidence to monitor a system's runtime behavior and investigate its failures (Yuan et al., 2012b; Yao et al., 2020). According to Lin et al. (2016), "engineers need to examine the recorded logs to gain insight into the failure, identify the problems, and perform troubleshooting." For this reason, it would be ideal to keep records of all evidence that can be analyzed (at runtime or later) to capture valuable information throughout the execution of software systems (Hassani et al., 2018). This evidence is generated through *logging*: the developers choose the points in the code where they add *log statements*, and these statements generate the *log entries* at runtime (Cândido et al., 2021; Li et al., 2018). The decision to generate more or fewer log entries, thus more or less evidence, depends on the choice of the log severity level (LSL).

As presented in Fig. 1, each log entry is usually composed of a timestamp, severity level, name of the software component involved in the event, and a log message. ***Severity levels*** indicate the degree of severity of the log message (Kim et al., 2020). For example, a less severe level is used to indicate that the system is behaving as expected, while a more severe level is used to indicate that a problem has occurred (Chen and Jiang, 2017a). A log entry will be added to the system log data every time the execution reaches a log statement whose severity level is equal to or exceeds that used to filter. These data can be parsed, processed, and stored to be consumed by monitoring activities. Operations engineers use them to monitor, for example, whether the system is functioning correctly, to analyze whether it is on the verge of failing, to identify behavioral anomalies, and to understand particularities during its operation through these data, or more generally, to understand how the system behaves (Cândido et al., 2021).

In system monitoring, system operators, having to deal with tracking various monitoring metrics, may receive a large volume of monitoring information. This includes many false warnings and alerts (Farshchi et al., 2018) and a high amount of noise, which affects log-based monitoring and diagnostics (Hassani et al., 2018; Li et al., 2017a; Rong

☆ Editor: W. Eric Wong.
* Corresponding author.
E-mail address: eduardo.mendes-de-oliveira1@uqac.ca (E. Mendes).

https://doi.org/10.1016/j.jss.2025.112643
Received 5 December 2024; Received in revised form 18 July 2025; Accepted 17 September 2025
Available online 24 September 2025

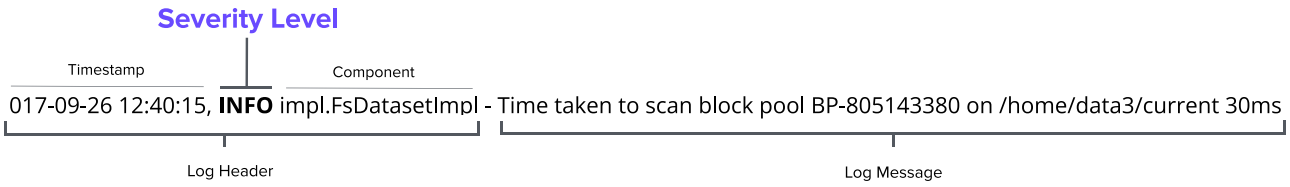

Fig. 1. Log severity level (LSL) in a log entry.
Source: Adapted from El-Masri et al. (2020).

et al., 2018). Part of this problem may come from the LSL chosen during development phase, which impacts the amount of log data that a software system produces (Lin et al., 2016; Chen and Jiang, 2017a; Chowdhury et al., 2018; Zeng et al., 2019). For example, if a system's logging verbosity is set to *Warn* level, only statements marked with *Warn* and higher levels (e.g., *Error*, *Fatal*) will be output (Chen and Jiang, 2017a).

In this sense, a software system with inappropriately chosen severity levels can produce fewer log entries than it should, or on the contrary, more log entries. If the choice results in too much logging, the system can produce two types of information in excess: *(i) unnecessary information:* the system produces log statements classified with a higher severity level than the semantics of its log messages (Zeng et al., 2019; Li et al., 2017a; Yuan et al., 2012a) and *(ii) redundant information:* the system produces repeated information arranged in different severity levels (Chen and Jiang, 2017a). On the other hand, if inappropriate severity levels result in too little logging, the system may hide critical information at lower severity levels (Hassani et al., 2018; Fu et al., 2014). This imbalance in log data will impact system performance (Chen and Jiang, 2017a; Li et al., 2017a; Yuan et al., 2012a) and maintenance (Li et al., 2017a; He et al., 2018), as well as affect log-based monitoring and diagnostics (Hassani et al., 2018; Li et al., 2017a; Rong et al., 2018).

Several studies propose solutions for the correct use of the LSL. Kim et al. (2020) propose an approach to verify the appropriateness of the LSL. Li et al. (2017a) propose a deep learning approach for LSL prediction using the logging locations. Li et al. (2020b) discuss where to apply logging locations and proposes a learning approach to provide code block logging suggestions. Other studies in the literature focus on *"where to log"* such as Zhao et al. (2017), Fu et al. (2014) and Li et al. (2020a). However, these works propose approaches that do not thoroughly explore adjustments related to LSL or provide detailed descriptions and categorizations.

Our research addresses this gap by proposing a comprehensive framework for understanding and applying severity levels. Unlike studies focusing on specific scenarios, our work broadly categorizes developers' intentions in changing LSL. We aim to examine LSL adjustments between system releases and explore methods to enhance LSL classification. To reach this goal, we investigate the following Research Questions (RQs):

- *RQ1: What log severity adjustments occur between system releases?*
- *RQ2: Why do severity level adjustments occur?*
- *RQ3: How can we improve LSL classification?*

To answer these questions, we use a three-phase methodology:

1. *a descriptive phase* to answer *RQ1*, which consists of examining the repositories of open-source Java systems to identify severity adjustments between different releases;
2. *an explanatory phase* to answer *RQ2*, which involves examining the issue descriptions and comments on the commits of these adjustments;
3. and finally, a *prescriptive phase* to answer *RQ3*, where we derive a set of heuristics for log severity selection based on our results.

Our results show that most LSL adjustments happen at the intersection of development and production environment logs. In many cases, developers may classify *Debug* messages as *Info* to enable debugging in production environments, where excessive logging could otherwise lead to issues. We also observed mutability in the LSL; the level may attenuate or even aggravate, depending on the maturity of the systems. Furthermore, the main guiding factors for the adjustments are the developer's experience and logging theory rather than formal guidelines. The results also show us that, in the absence of these formal guides, records in the issue reports of the systems act as an essential guide for understanding LSL.

The main contributions of this study are:

1. *A description of tendencies and patterns in severity adjustments:* Our study reveals significant trends, such as the predominance of experience-based adjustments and the changeability of severity levels with software maturity. These findings are fundamental to understanding how logs evolve in software development.
2. *A set of 24 heuristics to guide the choice, review, and adjustments of severity levels:* These heuristics are based on detailed analysis and provide practical guidance for developers and operations engineers to make informed decisions about log severity classification.

Paper structure. The remainder of this paper is organized as follows: In Section 2, we outline background information. We present our three-phase methodology in Section 3. In Section 4, we present the results of the Descriptive Phase (Phase 1), in Section 5, the results of the Explanatory Phase (Phase 2), and in Section 6, the results of the Prescriptive Phase (Phase 3). In Section 7, we discuss the results of all three phases. Related work is discussed in Section 8, and threats to validity are detailed in Section 9. Finally, conclusions and future work are presented in Section 10.

2. Background

2.1. Fundamental principles of log severity levels

In this section, we present the terms related to logging that are used in this study.

2.1.1. Log statement

A **log statement** is an instruction used in the source code to log a state, an event, or a behavior. Fig. 2 presents an example of a log instruction that records a message at the *Info* level.

2.1.2. Log entry

A **log entry** is the product of the execution of a log statement; the generated log entry can simply be displayed in the terminal during execution, sent to a log file, or transmitted over a data stream if a persistence strategy is used. Fig. 1 presents an example of a log entry.

```
249
250         LOG.info("Processing the event " + event.toString());
251
```

Fig. 2. Log statement of Hadoop (Apache, 2006).

Table 1
Selected logging libraries (ordered by the number of log severity levels (LSL)).

#	Library	Language	Levels
[L01]	Google Glog	C/C++	4
[L02]	Golang Glog	Golang	4
[L03]	OSLogging	Objective-C, Swifty	5
[L04]	Rust Lang	Rust	5
[L05]	Logback	Java	5
[L06]	SLF4J	Java	5
[L07]	Ruby Logger	Ruby	5
[L08]	LogLevel	JavaScript	5
[L09]	JS-logger	JavaScript	5
[L10]	CocoaLum	Objective-C	5
[L11]	Kotlin-logging	Kotlin	5
[L12]	SwiftyBeaver	Swifty	5
[L13]	Log4J	Java	6
[L14]	Commons Logging	Java	6
[L15]	Bunyan	JavaScript	6
[L16]	NLog[j]	C#	6
[L17]	Python Lang	Python	6
[L18]	ME Logging	.NET	6
[L19]	Serilog	.NET	6
[L20]	Log4PHP	PHP	6
[L21]	PinoJS	JavaScript	6
[L22]	Log.c	C/C++	6
[L23]	C-Logger	C/C++	6
[L24]	Zlog	C/C++	6
[L25]	Go-logging	Golang	6
[L26]	Log4m	MatLab	6
[L27]	Loguru	C/C++	6
[L28]	Spdlog	C/C++	6
[L29]	Java Util Logging	Java	7
[L30]	Logrus	Golang	7
[L31]	Uber-go/zap	Golang	7
[L32]	Bolterauer	VBA	7
[L33]	Swift-log	Swifty	7
[L34]	Loguru	Python	7
[L35]	Syslog-ng	C/C++	8
[L36]	PHP	PHP	8
[L37]	Monolog	PHP	8
[L38]	Winston	JavaScript	8
[L39]	Log4C	C/C++	9
[L40]	Log4Net	C#	15

2.1.3. Log severity levels

The **log severity levels (LSL)** indicates the degree of severity of the log message (Kim et al., 2020). For example, a less severe level is used to indicate that the system behaves as expected. In contrast, a more severe level is used to indicate that a problem has occurred (Chen and Jiang, 2017a).

To provide an overview of logging libraries and the LSL they encompass, the following section presents the results from a mapping of severity levels across various logging libraries, conducted in our previous study.

2.2. Logging library mapping

To better understand how severity levels are used in the logging libraries context, we mapped **40 logging libraries** covering 14 programming languages (Table 1) in a previous study (Mendes and Petrillo, 2021).

To find the appropriate logging libraries for these languages, we conducted a Google Search. We used only the first page of results for each language and obtained 160 hits, revealing more than 60 distinct libraries. We filtered the results by analyzing code and documentation repositories using the inclusion (IC) and exclusion (EC) criteria:

Table 2
All severity levels found in logging libraries (ordered by severity).

#	Level	Presence in libraries
1	Finest	3
2	Verbose	4
3	Finer	3
4	Trace	23
5	Debug	39
6	Basic	1
7	Fine	3
8	Config	1
9	Info	42
10	Success	1
11	Notice	10
12	Warn	41
13	Error	41
14	Fault	1
15	Severe	2
16	Critical	13
17	Alert	8
18	Fatal	22
19	Emergence	4

- **IC:** The library/language has a set of log severity levels;
- **EC1:** The library does not create log statements with LSL;
- **EC2:** The library is on GitHub and has less than 1,000 stars.

We found **19 different nomenclatures for severity levels** (Table 2), grouped in libraries with severity sets ranging from four (4) to fifteen (15) distinct levels, with 91% of the libraries having between five (5) and eight (8).

Of the 19 levels, a group of six severity levels are present in more than 50% of them, namely: *Info* (100%), *Warn* (98%), *Error* (98%), *Debug* (93%), *Trace* (55%), and *Fatal* (52%). However, these levels represent different degrees of severity depending on the library; for example, the most severe level in 48% of the libraries is Fatal, but in another 19%, it is *Error*.

The same variation occurs with less severe and intermediate levels, considering the variation in severity levels per library. For instance, one library uses one severity level for log debug instructions, while another library might use six different levels for the same task. Severity levels have associated numerical values in 38 of the 40 libraries, but three libraries show redundancy for levels with different names but equivalent semantics. Next, we present an overview of the definitions of the levels found in the libraries.

(a) Debug As defined by the libraries, the *Debug* level is characterized by detailed and low-priority or low-importance information, which is useful for debugging activities.

(b) Trace The *Trace* level is characterized similarly to *Debug*, yet it emphasizes an even lower priority than it.

(c) Info The *Info* level is associated with normal behavior, routine operations, and messages that describe the overall progress of the system. The level is also referenced as valuable for end users and system administrators.

(d) Warn The *Warn* level is typically described as indicating a dangerous situation, highlighting potential problems that might lead to a failure. It is often characterized as 'almost errors,' suggesting that while the application continues to run, something unexpected has occurred. This level is important for operators, end-users, or system managers as it indicates potential issues that may require attention.

(e) Notice The *Notice* level is similar to the *Info* level in several aspects, as it is used to describe normal events and highlight the application's progress. However, it also has a characteristic akin to the *Warn* level, as it can be used to describe potential failures, indicating a blend of both informational and warning functions.

Table 3
Severity level definitions (Mendes and Petrillo, 2021).

Severity level	Description
Debug	Describes variable states and details about interesting events and decision points in the execution flow of a software system, which helps developers to investigate internal system events.
Trace	Broadly tracks variable states and events in a software system.
Info	Describes normal events, which inform the expected progress and state of a software system.
Warn	Describes potentially dangerous situations caused by unexpected events and states. These must be observed, even if they do not interrupt the software system's execution.
Error	Describes the occurrence of unexpected behavior of a software system. These must be investigated, even if they do not interrupt the software system's execution.
Fatal	Describes critical events that bring a software system to failure.

(f) Error The *Error* level is typically described with terms such as 'major problem,' 'very serious error,' and 'unexpected conditions.' Additionally, it is noted that events registered at this level may or may not interrupt the application's operation. Even if they do not halt the application entirely, they can impede the progress of a specific request. In instances of logs at this level, it is crucial for an operator to be informed as soon as possible.

(g) Severe, Critical, Alert, Fatal, Emergency These five levels are associated with extremely severe error events in various library definitions. The *Severe*, *Critical*, *Alert*, *Fatal*, and *Emergency* levels each denote a very severe event. Specifically, the *Critical* level is often linked to disastrous occurrences and demands immediate attention. For the *Fatal* level, descriptions are more detailed, indicating situations where the application can no longer continue, may be forced to terminate prematurely, is on the verge of shutting down or stopping, might need to be aborted, or becomes completely unusable.

(h) Finest, Verbose, Finer, Fine These levels, found in a limited number of libraries, are described in comparative terms, focusing on the volume and detail of output, with an emphasis on their usefulness in debugging applications.

(i) Basic, Config, Success, Fault These four levels, unique to different libraries, include *Basic*, which is an alias for *Debug*, *Config* for detailing static configuration information useful in debugging, and *Success*, which is not specifically defined but numerically sits between *Info* and *Warn*. *Fault* is used to capture information about faults and bugs, without specified numerical values.

2.2.1. Log severity level convergence

The variation in the nomenclature of severity levels, the redundancy of levels for logging purposes and in numerical values led us to propose that the 19 severity levels can converge to higher levels of abstraction (Mendes and Petrillo, 2021). We identified a trend toward convergence into six severity levels, namely: *Trace, Debug, Info, Warn, Error,* and *Fatal* (Table 3), which can be categorized into four meta-levels, which we call *purposes for log severity levels*. The *Debugging Purpose* encompasses levels focusing on variable states and internal system behaviors. The *Informational Purpose* records expected software behavior. The *Warning Purpose* highlights issues for investigation without halting system execution. Lastly, the *Failure Purpose* encompasses the most severe levels dedicated to logging system failures (Table 4).

Table 4
Definitions of the purposes of log severity levels (Mendes and Petrillo, 2021).

Purpose	Description
Debugging Purpose	Describes levels used to log variable states and internal software system behavior events. It groups *Debug* and *Trace*, with *Trace* extrapolating *Debug*'s characteristics of describing variables and events.
Informational Purpose	Describes levels used to record the expected behavior of a software system.
Warning Purpose	Describes levels used to warn of unexpected software system behavior. It groups *Error* and *Warn* levels as both indicate issues (or potential issues) to be investigated but do not interrupt the system's execution.
Failure Purpose	Describes levels used to record software system failures.

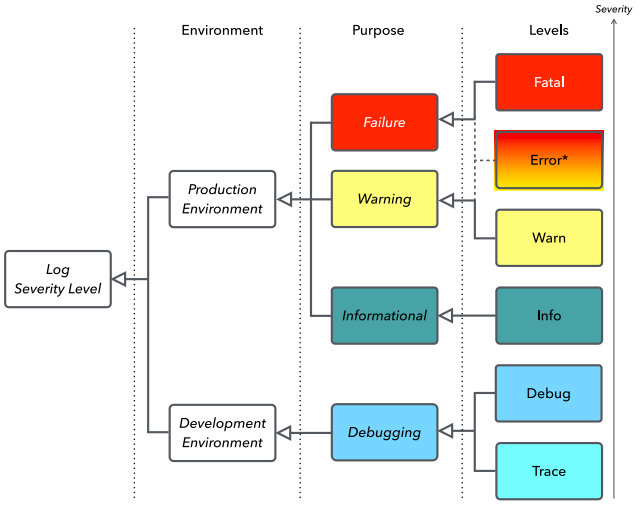

Fig. 3. Log severity level taxonomy: Six log severity levels are divided into two categories from left to right, first about environment and then about purpose. *Depending on the logging library, the *Error* level purpose classification varies between *Warning* and *Failure*.

Furthermore, log severity levels have two primary target environments: the development and production environments (Liu et al., 2019). While development tasks can benefit from log entries generated from all levels, the production environment requires only a subset of these entries. Notably, the *Info* level marks the threshold of the higher severity levels intended for the production environment. We present all these concepts about log severity level through a taxonomy in Fig. 3.

These log severity levels must form a total order in the mathematical sense, with each level considered strictly more severe than the previous one — in other words, there should not be two levels with the same severity. However, certain severity levels from different logging libraries may map to different purpose levels of our taxonomy.

2.3. Log severity level adjustments

In this work, we refer to a *log severity level adjustment* as a change that occurs in a log statement between two releases of the source code of an analyzed system.

This adjustment can be qualified with respect to the "direction" of the level change. If the change occurs from a less severe level to a more severe level, we shall call the adjustment an *aggravation*, *e.g.,*

Info to *Fatal*; if the change downgrades the level from a more severe to a less severe one, we shall call the adjustment an *attenuation*, *e.g., Warn* to *Debug*. For example, Listing 1 shows the same log statement in two different releases of HBase: the first from release Hbase/2.0.0RC0, and the second from release Hbase/2.5.0R0. In this case, there was a severity level adjustment, an aggravation, as *Info* is a higher severity level than *Debug*.

Listing 1: Severity level aggravation on Hbase

```
// Release Hbase/2.0.0RC0,
// TableAuthManager.java (line 188)
LOG.debug("Skipping permission cache refresh
          because writable data is empty")
// Release Hbase/2.5.0RC0, TableAuthManager.java (line
    136)
LOG.info("Skipping permission cache refresh
          because writable data is empty")
```

For an example of severity level attenuation, see Listing 2, where a log statement originally had a *Warn* level was changed to a *Debug* level in a later release.

Listing 2: Severity level attenuation on Hadoop

```
// Release Hadoop/release-0.15.3,
    PedingReplicationBlocks (line 186)
LOG.warn("PendingReplicationMonitor thread received
         exception. " + ie)

// Release Hadoop/release-0.16.0,
    PedingReplicationBlocks (line 187)
LOG.debug("PendingReplicationMonitor thread received
          exception. " + ie)
```

In addition to attenuation and aggravation categories, we refer to *"equivalences"* as a severity adjustment characterized by levels with different names but compatible severities, *e.g., Debug* to *Fine*; these episodes occur, for example, when the logging library changes from one release to another.

2.4. SLogAnalyzer

SLogAnalyzer is a tool developed in a previous work (M. Vasconcellos, 2023). SLogAnalyzer is designed to analyze the evolution of log code. It allows for the extraction and comparison of log statements and their semantic and syntactic characteristics in versions of open-source Java systems hosted on GitHub.

The tool performs repository cloning, copying all branches of the projects to be analyzed. After cloning the repositories, the SLogAnalyzer begins the process of extracting log statements. For each extracted statement, it detects a set of 21 pieces of information, including location (file, line, code snippet), severity level, message, and message without variables, when applicable. Additionally, it explores the source code to extract information about the methods in the release, such as location, number of log lines, cyclomatic complexity, and SLOC (Source Lines of Code).

2.4.1. Release comparison

The version comparison pipeline of the SLogAnalyzer is responsible for comparing files from different releases of the same project. It orders the releases chronologically and compares each sequential pair of releases in files common to both adjacent releases. For files present in both releases, the pipeline uses the Difflib library[1] to compare contents and identify changes. The similarity between log statements is measured by cosine similarity, which is commonly used to assess the similarity between texts (Singhal et al., 2001; Tan et al., 2005).

While several similarity measures exist, cosine similarity remains particularly well-suited for comparing log statements in our context. First, it is independent of vector magnitude, accommodating the variable-length nature of log messages across software versions (Turney and Pantel, 2010). Second, it effectively handles high-dimensional and sparse representations, which are typical of textual data (Plattel, 2014). Third, it focuses on relational patterns in vector space rather than exact structural matching (Turney and Pantel, 2010), which is relevant when detecting adjustments in log statements. These characteristics make it a particularly appropriate metric for identifying log statement modifications across software versions. In our implementation, changes with a cosine similarity above 0.5 are considered adjustments.

The objective of this pipeline is to compare data extracted from lines of code between consecutive versions of a project, categorizing changes into modifications, additions, and removals. At the end of the process, all information is stored in the database for further analysis. Refer to the work of M. Vasconcellos (2023) for more detailed information.

3. Study design

Previous research has already found adjustments in severity levels when comparing the source codes of different software versions (Anu et al., 2019; Li et al., 2017a). In our work, we want to deepen and broaden the knowledge about the adjustments and their causes, and in this way to improve the choice process of the log severity level. Our strategy uses a broad number of releases from the selected software to build an extensive view of the adjustments, using a 3-phase methodology (Fig. 4), preceded by an initial selection of projects (Phase 0).

In the *descriptive phase*, we examine the log severity level statically (source code) to paint an overview of the log severity level's distribution by release and the adjustments present in each selected software. In the *explanatory phase*, we examine the Jira's,[2] commit issue texts for each log severity level adjustment associated with the first phase to find the developers' explanations. In the *prescriptive phase* we propose heuristics based on the knowledge gained in the first two phases to guide the choice of severity.

3.1. Project selection (phase 0)

Below, we describe how we selected the projects for this work and the 3-phase methodology.

3.1.1. Criteria for project selection

To select the software systems for our study, we applied the following set of criteria:

- (IC1) There must be documented evidence in academic literature (journal or conference publications) of adjustments in the software's log severity levels;
- (IC2) The software must be open-source. Open-source software allows us to access the source code, publicly available documentation, and issue tracking.
- (IC3) The software's source code must be available on GitHub. This accessibility facilitates the extraction and analysis of log severity adjustments.

[1] https://docs.python.org/3/library/difflib.html

[2] Jira (https://www.atlassian.com/software/jira) is software from Atlassian that integrates with Git. It gives more context to commits, branches, tags, and pull requests.

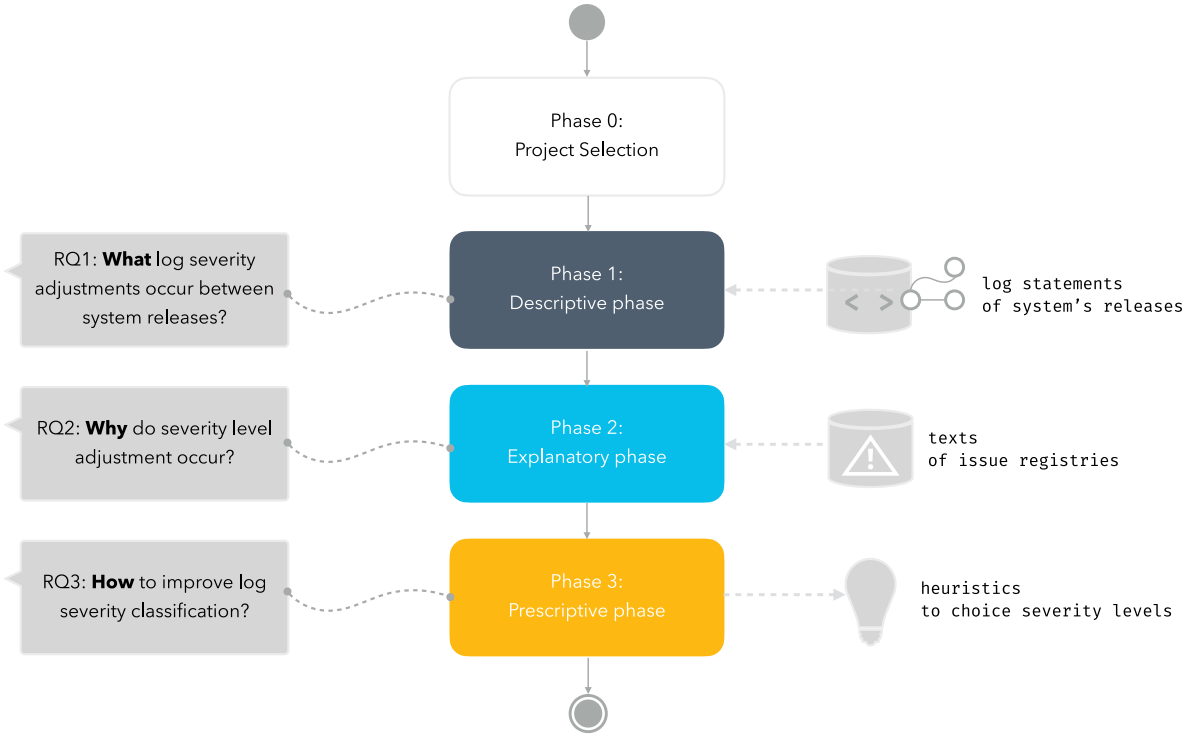

Fig. 4. Study Design.

- (IC4) The software must have issues registered on Jira[3]: Jira records detailed information about software issues, including developers' discussions about the commits.
- (EC1) Incompatibility of the software's log statements with the templates used by SLogAnalyzer. This is the tool we use to analyze log statements. If the software's log statements are incompatible with SLogAnalyzer's templates, performing a consistent and accurate analysis would be challenging.
- (EC2) Limited activity or sparse documentation of issues in the software's Jira repository: Active projects with well-documented issues are necessary to ensure a rich dataset for analysis. Projects with limited activity or sparse documentation would not provide sufficient data for a comprehensive study.

To meet the first inclusion criterion, we identified at least nine relevant studies that document adjustments in log severity levels (Shang et al., 2015; Chen and Jiang, 2017a,b; Zhao et al., 2017; Li et al., 2017b; Kabinna et al., 2018; Li et al., 2020a, 2021c; Zhang et al., 2022). Following the application of our selection criteria, approximately 60 software systems were found to meet the initial inclusion conditions. However, due to the resource-intensive nature of manual curation and analysis, it was not feasible to evaluate all of these systems. As a result, we made a selective choice based largely on the impact and popularity of the projects, ultimately focusing on three widely-used open-source systems for detailed analysis. Next, we present each of the projects selected for our study.

[3] The selected projects use Jira as their issue tracking system, which is common for many open-source and commercial projects on GitHub. This integration allows for advanced project management features that Jira offers. Therefore, only Jira issues were considered in our study.

Hadoop. The Apache Hadoop software library provides a framework for processing large data sets across computer clusters using straightforward programming paradigms. Designed to extend from individual servers to thousands, it emphasizes local computation and storage. Instead of depending on high-availability hardware, Hadoop is engineered to manage failures at the application layer, ensuring a resilient service even with potentially unreliable computers.[4]

HBase. Apache HBase is a distributed, versioned, non-relational database that offers real-time, random read/write access to Big Data. The project aims to host extensive tables – encompassing billions of rows and millions of columns – on clusters of commodity hardware. Modeled after Google's Bigtable, HBase provides Bigtable-like capabilities using Hadoop and HDFS as its storage infrastructure.[5]

Kafka. Apache Kafka is an open-source platform specializing in distributed event streaming. Numerous companies utilize it for data pipelines, streaming analytics, data integration, and vital applications.[6]

3.2. Descriptive phase (phase 1)

This phase aims to describe the adjustments in log severity between system releases. To achieve this, we obtained log data from adjacent releases and compared them. The outlined process can be seen in Fig. 5, and we will provide a detailed description of it below.

We analyzed only stable releases and discarded those designated as release candidates. Table 5 shows the selected systems and the number of versions and releases that are part of this study.[7]

[4] https://hadoop.apache.org/
[5] https://hbase.apache.org/
[6] https://kafka.apache.org/
[7] In this paper, we use the terms *version* and *release* as follows: *Release* is part of a *Version*: e.g., system release 2.1.2 is a release of version 2.*, just as release 3.3.4 is a release of version 3.*.

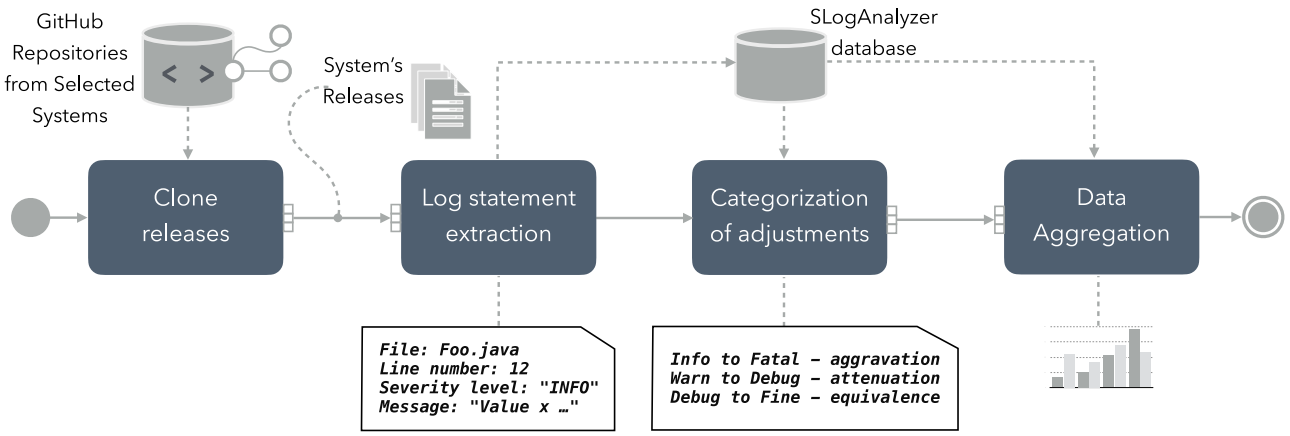

Fig. 5. Overview of the descriptive phase (Phase 1). This phase analyzes log severity level (LSL) adjustments between adjacent software releases. The process starts with SLogAnalyzer cloning the releases of selected open-source systems from GitHub. It then extracts log statements, capturing information such as severity level, file name, line number, and log message. These statements are compared across releases to detect LSL adjustments, which are categorized as aggravation, attenuation, or equivalence, and assigned a severity degree. Finally, all data is aggregated to support quantitative analysis of adjustment patterns across systems.

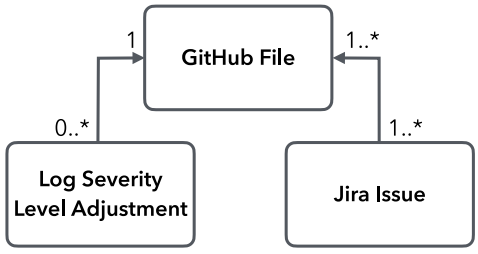

Fig. 6. Relationship between *Adjustments*, *GitHub Files*, and *Jira Issues*: A file on GitHub may contain adjustments and can be associated with one or more issues in Jira. Conversely, a Jira issue is linked to one or more files on GitHub.

Table 5
Selected open-source systems.

#	System	Versions	Releases	Logging Library
[S1]	Hadoop	4 - [0.*, 1.*, 2.*, 3.*]	131	Commons Logging, SLF4J + Log4J
[S2]	Kafka	4 - [0.*, 1.*, 2.*, 3.*]	60	Log4J
[S3]	HBase	3 - [0.*, 1.*, 2.*]	204	Log4J
	Total	11	395	

First, we cloned the releases of the chosen systems using SLogAnalyzer (M. Vasconcellos, 2023). We retrieved all files from each release on GitHub to extract the code related to log statements. This extraction included details such as the *severity level, message, file name, line number, a surrounding code snippet* for context, and other satellite information. Focusing on log statements, we use the SLogAnalyzer to detect LSL adjustments (aggravations, attenuations, and equivalences) between different software system releases. We also added a degree to each adjustment representing the distance from the *initial* to the *final severity level*; equivalence adjustments receive degree 0 (zero) (see Table 6).

Using the data automatically processed through SLogAnalyzer, we manually aggregated the adjustments' data. This processing further allowed us to quantify and understand the distribution of these adjustments across the various releases analyzed. We have also compiled a summary of metrics for each release, detailing the total number of log statements, files affected by adjustments, the unique messages, and the distribution of log severity levels (see Table 1).

Table 6
Explanation of LSL adjustments: "$>$" denotes adjustment to a lower severity level, "$<$" denotes adjustment to a higher severity level, and "$\equiv$" denotes adjustment to an equivalent severity level.

Category	Adjustment examples	Degree
Attenuation	Debug > Trace	1
	Info > Trace	2
	Fatal > Info	3
Equivalence	Fine ≡ Debug	0
	Finer ≡ Debug	0
	Warning ≡ Warn	0
Aggravation	Trace < Debug	1
	Debug < Warn	2
	Info < Fatal	3

3.3. Explanatory phase (phase 2)

Based on the LSL adjustments obtained in the previous phase, we analyzed the Jira issue texts associated with these adjustments to investigate and understand their causes. The relationship between the adjustments, GitHub files, and Jira issues is depicted in Fig. 6.

3.3.1. Jira issues selection

In Fig. 7, we summarize the steps for selecting Jira issues to investigate, and below, we describe each of them.

From the SLogAnalyzer database, using manual SQL queries,[8] we filtered the files from each selected software system to obtain the set corresponding to the LSL adjustments. Then, using custom scripts based on Puppeteer,[9] a JavaScript library, we automatically retrieved the corresponding *Jira issue ID* and associated *link* (URL) from the GitHub file page, provided they were available.

In the following step, we engaged the Jira API,[10] to glean the *summary* (the issue title), *summary relevance description*, and *comments*

[8] Available in our reproducibility kit: https://doi.org/10.6084/m9.figshare.26253776.v2
[9] https://pptr.dev/
[10] https://developer.atlassian.com/cloud/jira/software/rest/

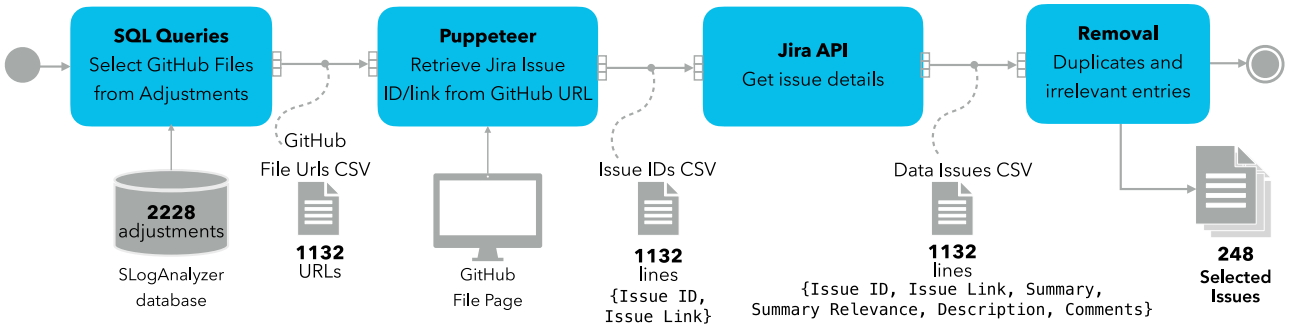

Fig. 7. Process flow for selecting Jira issues to investigate (Explanatory Phase — Phase 2). Starting from 2228 log severity level (LSL) adjustments identified in Phase 1, we used SQL queries to retrieve the corresponding GitHub file URLs (1132 in total). Using Puppeteer, we extracted Jira issue IDs and links from the GitHub pages. Then, the Jira API was employed to collect issue metadata, including summaries, descriptions, and comments. A keyword-based regular expression was applied to the comments to identify potential relevance to LSL adjustments. Finally, duplicate and irrelevant entries were manually filtered, resulting in 248 Jira issues selected for qualitative analysis.

for each issue. We collected only resolved issues. The *summary relevance* field is derived by applying a regular expression to the *comments* field to determine if the issue contains any evidence of LSL adjustments. The regular expression used in this step is ‘‘`(fine|trace|debug|info|warn|error|fatal |logging| log | logs | logger |log level|severity level|change level|change severity|slf4j|log4j|logback |noisy| verbose|spammy|overload|fail)`’’:

- *“logging”, “log”, “logs”, “logger”, “log level”,* and *“severity level”* are common words that may be present in discussions about logging.
- Keywords like *“fine”, “trace”, “debug”, “info”, “warn”, “error”, and “fatal”* are also included to match with the severity levels.
- Keywords to flag the log library names used by the systems are *“slf4j”, “log4j”,* and *“logback”.*
- We also include adjustment-related expressions such as *“change level”,* and *“change severity”.*
- Keywords like *“noisy”, “verbose”, “spammy”, “overload”,* and *“fail”* represent common log issue words (Li et al., 2020a; Yang et al., 2021).

Upon collecting the comments, we underwent a data cleanup process that included extracting text markup tags (*e.g.,* `{color}, {noformat}, {panel}`).

The final filtering and validation of relevant issues were conducted manually. All tagging, interpretation of issue content, and the derivation of severity adjustment motivations were performed manually as part of our qualitative analysis.

3.3.2. Jira issues analysis

At this stage, we wanted to confirm that the developers’ discussion focused on adjusting the severity level. This would enable us to locate adjustment explanations, justifications for previous level assignments, and potential heuristics to incorporate into Phase 3.

To perform this analysis, for each of the selected issues, we (i) examined the summary, description, and comments, (ii) summarized the content of the issue, and (iii) added tags to classify the issue text relationship with the LSL adjustment. We tagged the issues using a systematic process inspired by the *keywording* method of Petersen et al. (2008). This process aims to create a classification framework that separates the adjustments according to their focus. The process followed the following steps:

1. Identification of tags and explanations. The first author (R1) examined each Hadoop issue, identifying explanations for adjustments, potential heuristics, and assigning tags. After that, R1 presented 20% of the results to the second author (R2) to create a shared understanding of the process. Next, individually, R1 examined the remaining HBase and Kafka issues, and R2 examined the issues from the three projects. The results of the two authors were discussed to reach a consensus on the analyses. The fourth author (R4) evaluated the results and resolved conflicts between non-consensual results. Although we were inspired by the keywording method proposed by Petersen et al. (2008), the initial tags and adjustment explanations were not predefined. Instead, they were inductively derived from developers’ justifications in the issue reports. This allowed us to surface a wide variety of motivations for log severity changes. Example of tags and explanations are:

- Tags: *“Not related to the adjustment”, “Does not explain the reason for the adjustment”, “Verbose ERROR”, “Event handled by client”, “Masking DEBUG into an INFO”, “Not affect the normal service”.*
- Explanations: *“Info causing a lot of noise”, “It can probably be changed to log at DEBUG level instead”, “it doesn’t take any action but logged in error mode”, “Moving logging APIs over to slf4j in hadoop-mapreduce-client-app”.*
  A complete list of the 45 initial tags used for organizing and analyzing severity adjustment motivations is provided in Appendix A.

2. Determining the main categories. To derive the five main categories, we followed an approach to iteratively refine and group tags and explanations. The steps included:

- R1 identified tags and explanations with greater frequency and relevance for severity level adjustments and proposed initial categories. The initial categories are: *“Bad adjustments”, “Contextual adaptation”, “Extraordinary exceptions”,“Historical relevance of logs”, “Inconsistencies in project practice”, “Severity levels not appropriate for the situation”, “Silent Problems”, “Universal rules”, “Verbose log”, “Wrong classification”.* These initial categories emerged from observed patterns across the dataset and served as an intermediary step in the construction of the five main categories presented in Section 5.2.
- Refinement through discussion: Authors R1 and R4 grouped the results, identifying similarities and the main trends in adjusting severity levels against the initial categories. For example, most of the adjustments related to repetitive log entries were motivated more by the realization of the enormous amount of data produced by the system (“Verbose log”) than the content generated by the log statement. In cases of adjustments like this, based on intuition and practice, we grouped them under the Experience category.

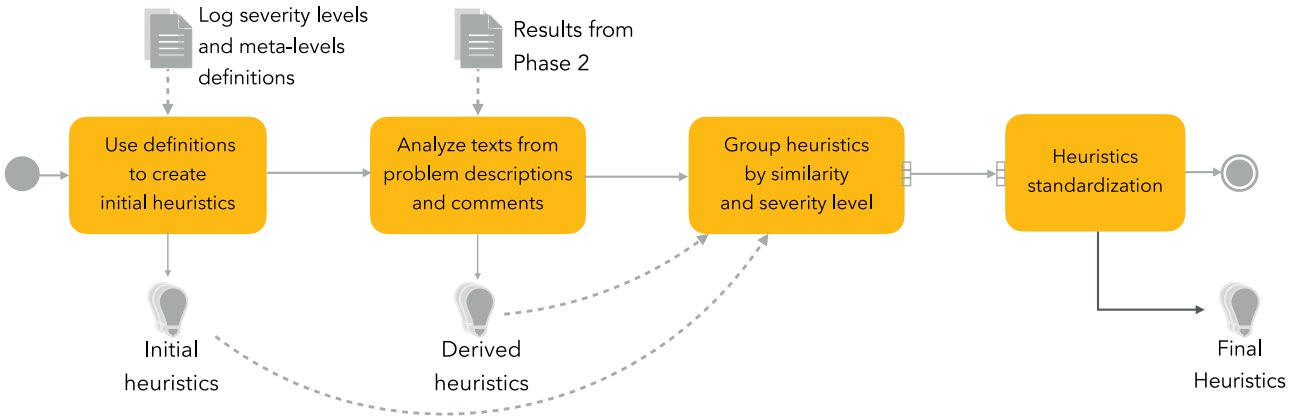

Fig. 8. Prescriptive phase process: deriving and standardizing LSL heuristics.

This iterative process helped merge overlapping themes and split broad groups into more specific categories where necessary.
- Validation against issues: R1 and R2 validated each category by mapping it against the relevant issues to ensure that it accurately represented the underlying reasons for severity adjustments.

At the end of this analysis, we obtained a set of Jira issues tagged according to their relevance to the severity adjustment and an overview of the main reasons behind these adjustments.

3.4. Prescriptive phase (phase 3)

After describing and exploring the LSL adjustments in Phases 1 and 2, during the prescriptive phase, we employ, in the prescriptive phase (Fig. 8), an approach inspired by Grounded Theory (Glaser and Strauss, 2017) to extract and formalize heuristics related to LSL. We use data from two primary sources:

- The fundamental principles of log severity levels, produced in our previous work (Mendes and Petrillo, 2021), as presented in the background section.
- The results obtained in Phase 2 of the current work, which aimed to understand the reasons for severity adjustments at the log level.

3.4.1. Generation of heuristics

Initially, three authors (R1, R2, and R4) independently reviewed the results obtained in Phase 2, noting down potential patterns or rules that could serve as heuristics. These patterns were mainly derived from the recurring themes in the problem descriptions and comments, as well as the researchers' understanding of the severity level definitions. The steps in this process are described below:

Step 1: Initial heuristic generation (open coding). Based on the previously established definitions of severity levels and purposes (Tables 3 and 4), we began the process of heuristic generation.

Step 2: Analysis of incident texts (open coding).

(a) *Identification of potential heuristics.* In this step, we extracted potential heuristics (PH) from Phase 2 based on the analyzed issues, when available. Generally, we looked for imperative statements conveying best practices that could be associated with patterns observed in other issues.

(b) *Recording preliminary observations.* Using the results from the categorization of adjustments in Phase 2, authors R1 and R2 recorded general preliminary observations about LSL adjustments as learnings from this phase. Each preliminary observation (PO) was linked to the related issues. For instance:

- [PO1]: *"There are instances of Debug to Info adjustments exclusively to facilitate the debugging process"*. (MAPREDUCE-5766, HDFS-14759, FIO-1839, MAPREDUCE-3692)

Step 3: Grouping heuristics by similarity and severity level.

(a) *Collaborative analysis and initial heuristic elaboration* (axial coding): Next, authors R1, R2, and R4 met to discuss their findings. Each PH and PO were reviewed, and potential heuristics were designed based on their combination. Differences in the identified heuristics were debated, and through a consensus approach, the team narrowed it down to a list of initial heuristics.

(b) *Refinement and validation* (selective coding): We further refined the initial and derived heuristics by matching them to the associated incidents and preliminary observations to ensure that multiple instances supported each heuristic. Statistical descriptions of PH, PO, and incidents were used as indicators of each heuristic's validity. For example, H9, which states that *"Important information is unsuitable for Debugging Purpose severity levels"*, is supported by PH10 and PO5 and linked to several issues such as HADOOP-12789 and HADOOP-1034. At this stage, we also eliminated redundant heuristics.

Step 4: Heuristic standardization.

(a) *Transition to impersonal form.* Initially, heuristics were formulated based on observations and identified patterns. These initial formulations were not standardized and were sometimes expressed in imperative form. Subsequently, each heuristic was revised to be presented impersonally, aiming to reinforce objectivity and neutrality in the recommendations.

(b) *Conformity to RFC 2119.*[11] Finally, each heuristic was aligned with the terms of RFC 2119 (Bradner, 1997), ensuring that the expressed requirements were both clear and suitable for practical application.

(c) *Final review and documentation.* The final set of heuristics was reviewed and documented. The frequency of issues related to each heuristic was also observed, providing an overview of its empirical basis.

[11] RFC 2119 provides precise terminology to express requirements in a technical context, which is essential to ensure that recommendations are correctly interpreted. Words such as MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY were used to indicate the level of obligation or recommendation associated with each heuristic (Bradner, 1997).

Table 7
System extractions.

System	Number of files	Number of distinct messages	Number of statements
Hadoop	2701	16,877	32,046
Hbase	1568	8755	17,278
Kafka	410	2769	5557
Total	4679	28,401	54,881

This iterative and collaborative approach contributed to the heuristics being based on our study's quantitative and qualitative data.

3.5. Toward results

Next sections present the results obtained from our comprehensive analysis of LSL adjustments across multiple software releases. Leveraging a 3-phase methodology, we delve into how severity levels shift over software versions, shedding light on patterns and their implications.

4. Descriptive phase (phase 1) | results

This section describes the quantitative results of our analysis of the log statements across 395 releases of the selected projects. In this phase, we analyzed more than 54,000 log statements distributed in almost 5000 source files (Table 7) to establish a panorama of log severity levels and their adjustments. We present the distribution of severity levels throughout the releases, detailed data on severity adjustments, and totals for the categories and each adjustment found.

4.1. Log severity level distribution by release

The following charts (Figs. 9, 10, and 11) provide an overview of how the LSL distribution in analyzed systems has changed over time as percentages. These percentages represent the proportion of each severity level relative to the total number of log entries for each release. For instance, a 40% value for the *Debug* severity level in a particular release means that 40% of all log entries for that release were classified as *Debug*. They show the frequency of each severity level across selected releases, which helps understand patterns and trends. The main points are:

- Log statements classified with the *Info* severity level are prevalent in most releases, highlighting their role in recording the standard behavior of systems. The exception is in Kafka releases, where the frequency of Info severity is similar to that of the *Debug* logs;
- *Debug* log statements come in second place and suggest the importance of logging as a debugging tool for developers;
- In third place, *Warn* log statements record a relatively constant occurrence, suggesting a certain regularity in problems or events requiring attention;
- The *Error* level log is the most severe level in the most recent releases of the three systems. In Hadoop and HBase, the log statements that use this level form the fourth largest group, while in Kafka it is the third. Looking at the first two systems, this suggests that the systems are healthy. In Kafka, on the other hand, this may suggest a higher number of behaviors susceptible to failure;
- *Trace*, while less prevalent in Hadoop and HBase, is more prominent in Kafka releases than the other systems;
- *Fatal* logs are the least present LSL, which may suggest the integrity of system operations, or that the log library used does not have this severity level. This is the case with Hadoop, which uses SLF4J, featuring 5 levels of severity, not including *Fatal*,[12] as a facade for the Log4J library. Previously, the facade was Commons Logging which has 6 severity levels, including *Fatal*.

[12] www.slf4j.org/api/src-html/org/slf4j/spi/LocationAwareLogger.html

Table 8
Adjustment categories on severity level.

#	Category	Adjustment	Degree	Total
1	Attenuation	Debug > Trace	1	295
2		Info > Trace	2	57
3		Info > Debug	1	463
4		Info > Fine	1	2
5		Warn > Trace	3	10
6		Warn > Debug	2	93
7		Warn > Info	2	64
8		Error > Trace	4	4
9		Error > Debug	3	46
10		Error > Info	2	41
11		Error > Warn	1	66
12		Fatal > Info	3	1
13		Fatal > Warn	2	7
14		Fatal > Error	1	184
15	Equivalence	Fine ≡ Debug	0	55
16		Finer ≡ Debug	0	16
17		Finest ≡ Trace	0	2
18		Warning ≡ Warn	0	51
19		Severe ≡ Fatal	0	11
20	Aggravation	Trace < Debug	1	133
21		Trace < Info	2	24
22		Trace < Warn	3	9
23		Trace < Error	4	3
24		Debug < Info	1	282
25		Debug < Warn	2	69
26		Debug < Error	3	29
27		Fine < Info	1	7
28		Finer < Info	1	3
29		Info < Warn	2	74
30		Info < Warning	1	2
31		Info < Error	2	48
32		Info < Fatal	3	3
33		Warn < Error	1	52
34		Warn < Fatal	2	7
35		Error < Fatal	1	15
			Total	2,228

4.2. Log severity level adjustments overview

The analysis of LSL adjustments reveals the dynamic nature of logging practices across different software releases. We categorize these adjustments into three main types: *aggravation*, *attenuation*, and *equivalence*. Each category reflects a strategic choice by developers, impacting the log's informativeness and diagnostics. Below, we provide an analysis presenting the proportion and impact of each type of adjustment on logging practices.

We have identified **35 types of adjustments** in total: 14 attenuation types, 16 aggravation types, and 5 equivalence types (Table 8). As we analyzed the data, we found **more instances of attenuation than aggravation.** Specifically, out of the 2228 adjustments we examined, we saw 1333 occurrences of attenuation (roughly 60%), 760 occurrences of aggravation (about 34%), and 135 occurrences of equivalence (roughly 6%). Regarding the adjustment degree, 1-degree adjustments make up 67.50% of the total, while 2-degree adjustments account for 21.72% (Fig. 13).

The primary adjustments in log levels, particularly in terms of attenuation and aggravation, are notably prevalent between the *Debug - Trace* and *Info - Debug* levels, as illustrated in Fig. 12. These adjustments are characterized by a degree of 1.

4.2.1. Hadoop

We evaluated 130 releases from four sets of Hadoop versions: 67 from version 0.*, 10 from version 1.*, 33 from version 2.*, and 20 from version 3.* (Table 9).

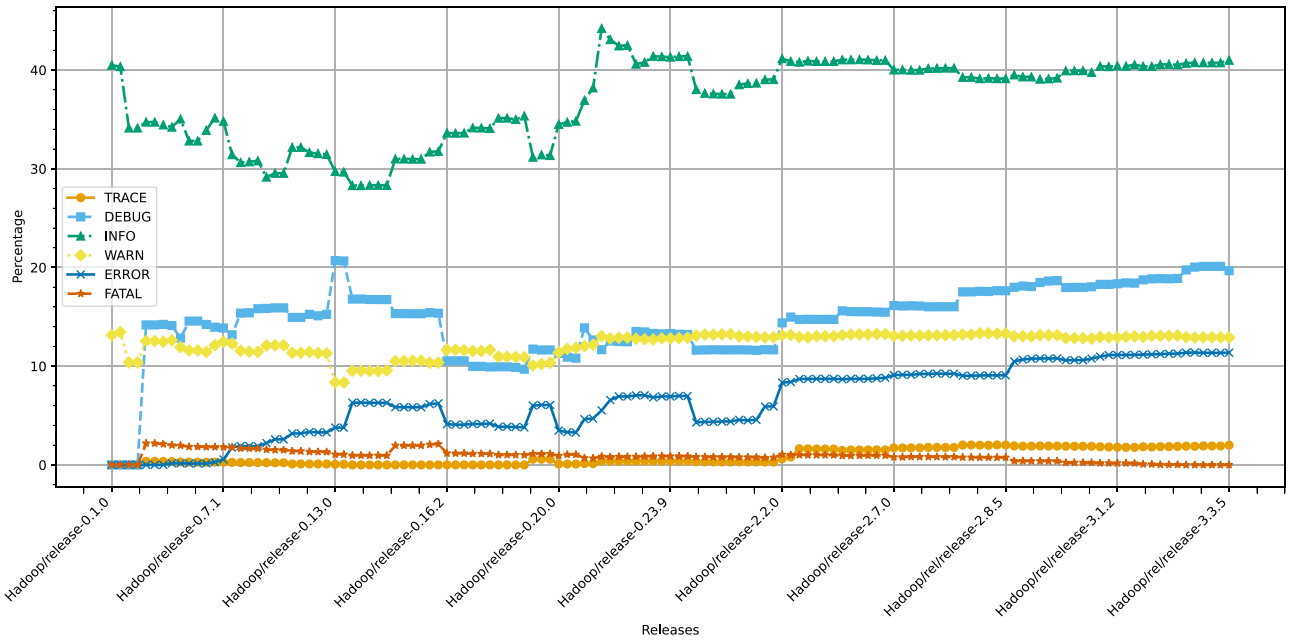

Fig. 9. Log severity level (LSL) distribution Hadoop: Data from all 130 Hadoop releases, with 11 labels shown for clarity.

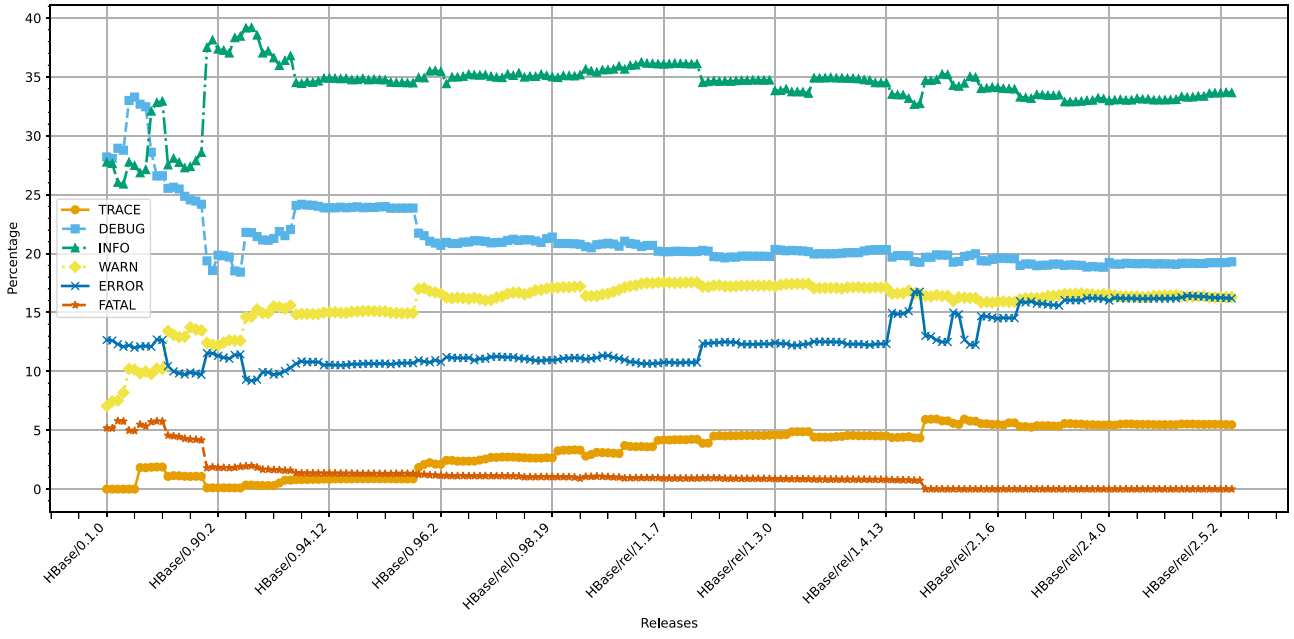

Fig. 10. Log severity level (LSL) distribution HBase. Data from all 203 HBase releases, with 11 labels shown for clarity.

The 1037 Hadoop adjustments are 587 attenuations, 315 aggravations, and 135 equivalences. The equivalences are mainly due to a change in the logging library.

Looking at Fig. 14, we mostly noticed attenuation adjustments in the LSL as Hadoop versions evolved.

The adjustments with the most occurrences are *Info* > *Debug* (1-degree, 303 occurrences), *Fatal* > *Error* (1-degree, 121 occurrences), and *Debug* < *Info* (1-degree, 146 occurrences) (Fig. 15). The *Fatal* > *Error* adjustment (1-degree) occurs only in versions 2.* and 3.*.

4.2.2. HBase

We evaluated 203 releases from three groups of HBase versions: 89 from versions 0.*, 58 from versions 1.*, and 56 from versions 2.* (Table 10).

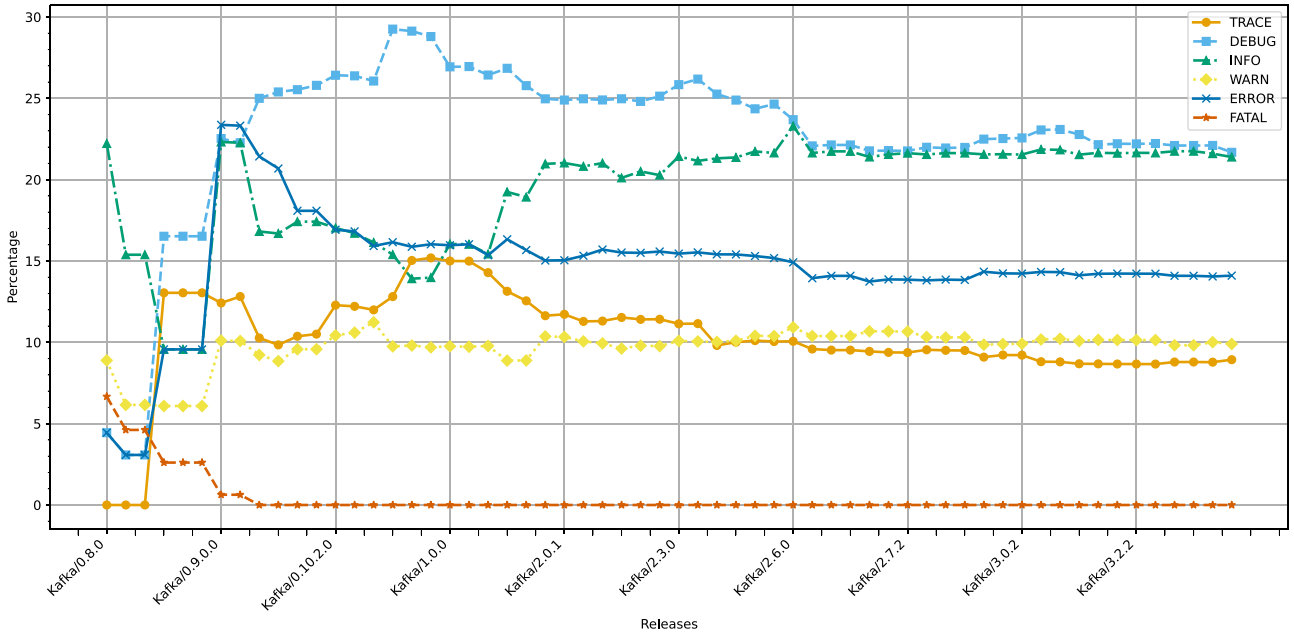

Fig. 11. Log severity level (LSL) distribution Kafka. Data from all 49 Kafka releases, with 10 labels shown for clarity.

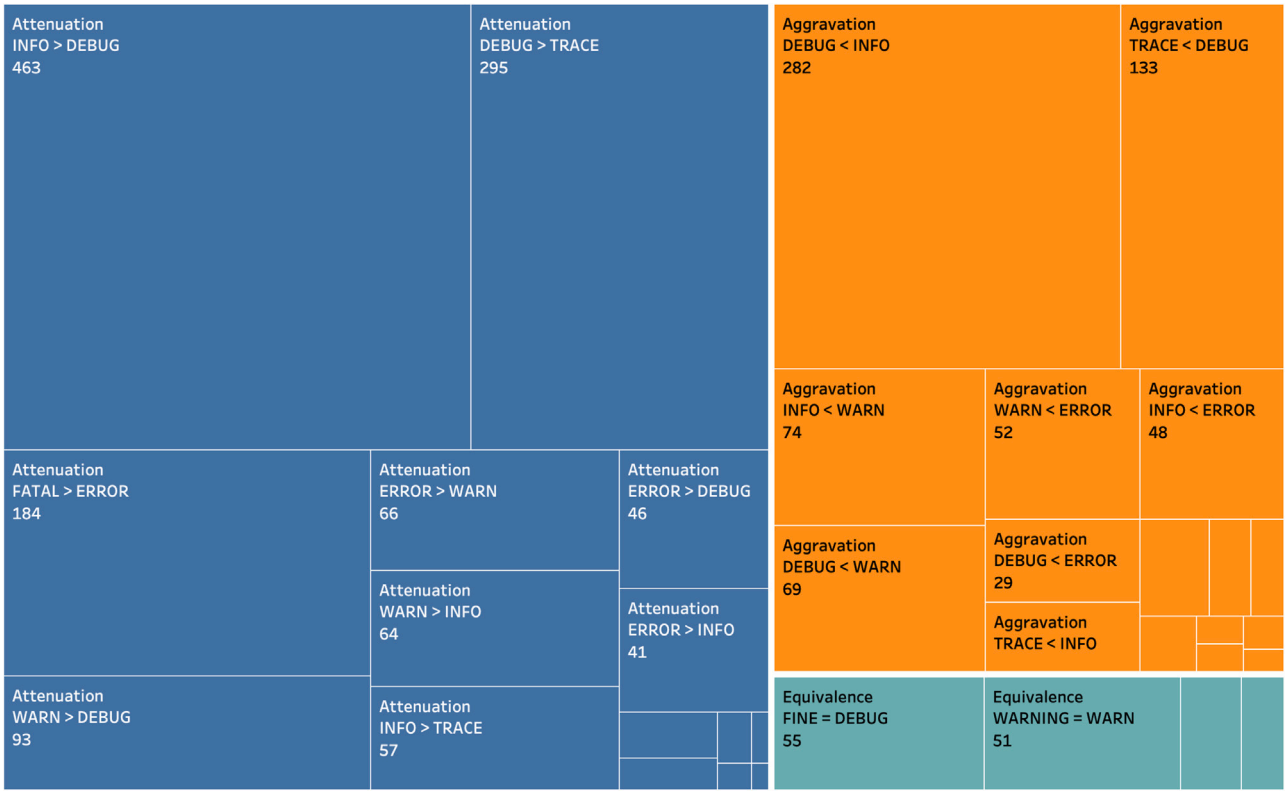

Fig. 12. Adjustments to log levels on selected systems.

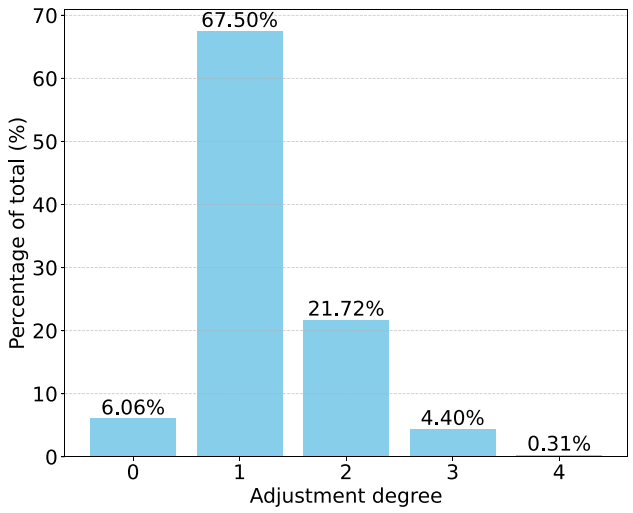

Fig. 13. Adjustment percentage by degree.

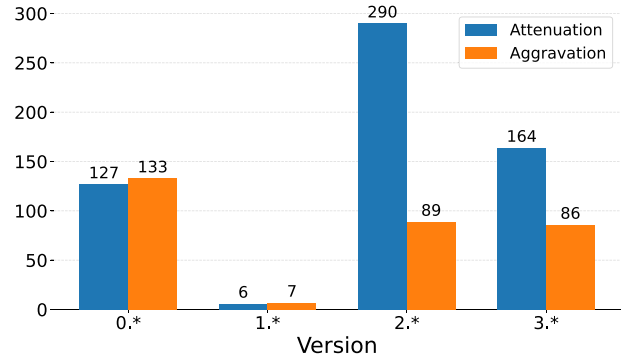

Fig. 14. Hadoop: Attenuation and aggravation.

Table 9
Adjustments on Hadoop.

Version	Releases	Adjustments	Attenuation	Equivalence	Aggravation
0.*	67	395	127	135	133
1.*	10	13	6	0	7
2.*	33	379	290	0	89
3.*	20	250	164	0	86
Total	**130**	**1037**	**587**	**135**	**315**

The 972 HBase adjustments are 637 attenuations, 335 aggravations, and 0 equivalences.

Looking at Fig. 16, we notice that in all versions there are more attenuation than aggravation in LSL adjustments.

The adjustments with the most occurrences are *Debug* > *Trace* (1-degree, 259 occurrences), *Info* > *Debug* (1-degree, 117 occurrences), *Trace* < *Debug* (1-degree, 96 occurrences), *Debug* < *Info* (1-degree, 90 occurrences), and *Fatal* > *Error* (1-degree, 63 occurrences) (Fig. 17).

4.2.3. Kafka

We evaluated 49 releases from four sets of Kafka versions: 7 from version *0.**, 5 from version *1.**, 23 from version *2.**, and 14 from version *3.** (Table 12).

The 219 Kafka adjustments are 109 attenuations, 110 aggravations, and 0 equivalences.

Table 10
Adjustments on HBase.

Version	Releases	Adjustments	Attenuation	Equivalent	Aggravation
0.*	89	330	224	0	106
1.*	58	271	178	0	93
2.*	56	371	235	0	136
Total	**203**	**972**	**637**	**0**	**335**

Table 11
Explanatory phase extraction numbers.

Metrics	Hadoop	HBase	Kafka	Total
Adjustments	1037	972	219	2228
Files with Adjustments	502	515	115	1132
Files with an Associated Jira Issue	483	514	99	1096
Files without an Associated Jira Issue	19	1	9	29
Jira Issues (without duplicates)	366	326	85	776
Jira Issues to Investigate	135	94	19	248
Adjustment-related Jira Issues	77	36	6	119

Table 12
Adjustments on Kafka.

Version	Releases	Adjustments	Attenuation	Equivalence	Aggravation
0.*	7	62	52	0	10
1.*	5	22	4	0	18
2.*	23	103	42	0	61
3.*	14	32	11	0	21
Total	**49**	**219**	**109**	0	**110**

Looking at Fig. 18, we notice that in the first set of versions (0.*), there are more attenuations and in the last three sets of versions, there are more aggravations adjustments to the LSL.

The adjustments with the most occurrences are *Debug* < *Info* (1-degree, 46 occurrences), *Info* > *Debug* (1-degree, 43 occurrences), *Debug* > *Trace* (1-degree, 30 occurrences), and *Trace* < *Debug* (1-degree, 29 occurrences) (Fig. 19).

5. Explanatory phase (phase 2) | results

In this section, we present the results of the Explanatory Phase of our study, focusing on the reasons behind severity level adjustments in the log records of Hadoop, HBase, and Kafka. First, we present the quantitative data analyzed, starting from the adjustments identified in Phase 1 to the issues that effectively contributed data to the study.

The topics are organized into five main categories: Fundamental Principles Adjustments (§ 5.3), Historical Practice Adjustments (§ 5.4), Experience-Driven Adjustments (§ 5.5), Logging Library Swap Adjustments (§ 5.6), and Guideline-Based Adjustments (§ 5.7). Each category is supported by specific issue reports illustrating the rationale behind these adjustments, providing an overview of the diverse factors influencing logging practices.

The final subsection synthesizes insights from the analyzed data, proposing preliminary observations and potential heuristics that could guide future log severity adjustments.

5.1. Data overview

From the 2228 adjustments in Phase 1, we obtained 1132 files across the selected systems, totaling 502 for Hadoop, 515 for HBase, and 115 for Kafka. When searching for the issues associated with these

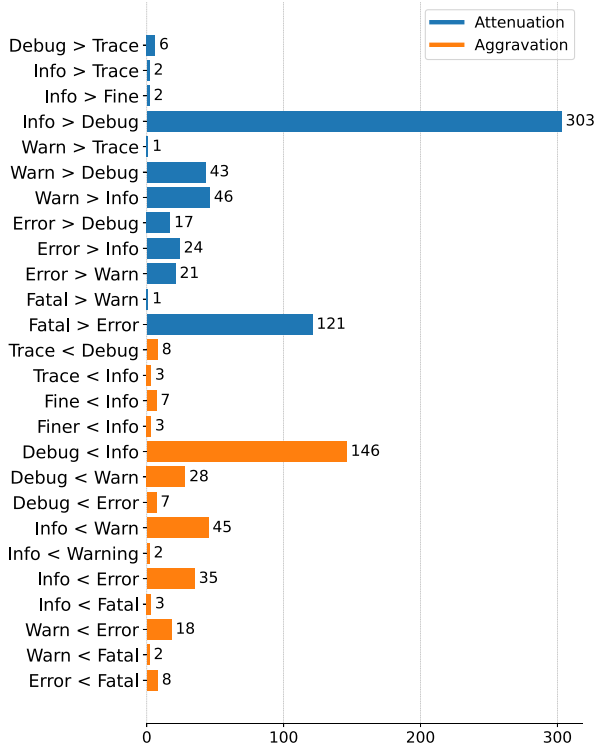

Fig. 15. Hadoop: Adjustment occurrences.

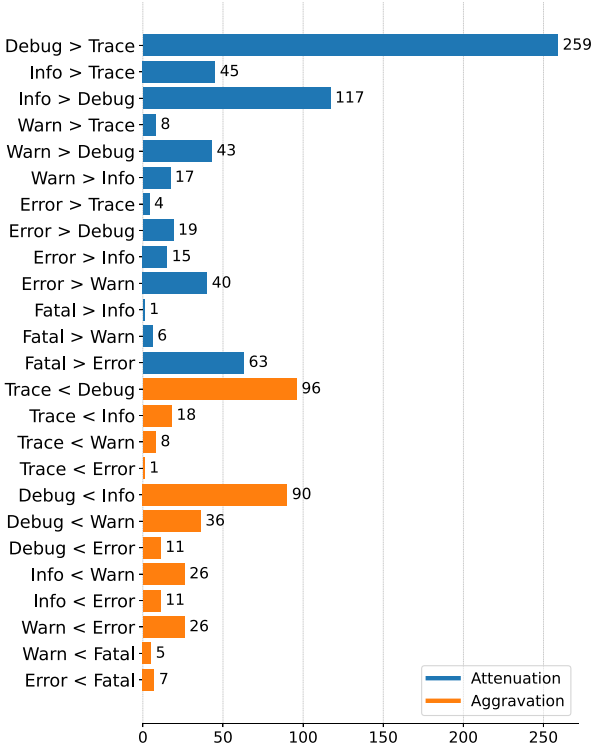

Fig. 17. HBase: Adjustment occurrences.

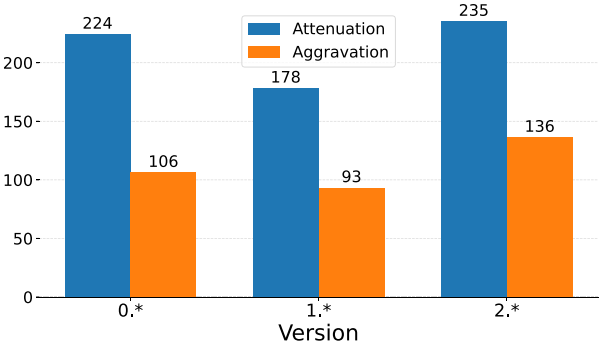

Fig. 16. HBase: Attenuation and aggravation.

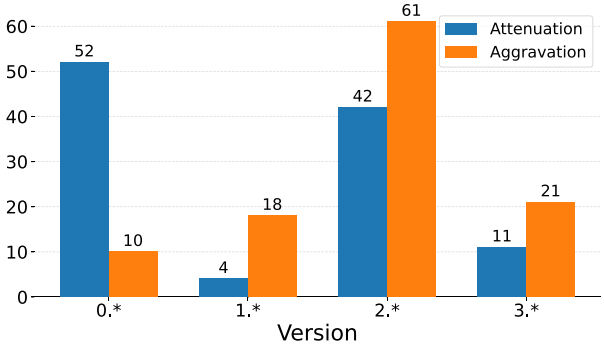

Fig. 18. Kafka: Attenuation and aggravation.

files, removing duplicates and irrelevant entries, we got 248 Jira issues — 135 for Hadoop, 94 for HBase, and 19 for Kafka. After manual analysis, excluding issues unrelated to the adjustment or when they were, did not explain the reason for the adjustment, our final set consists of 119 adjustment-related issue reports, with 77 from Hadoop (Appendix B), 36 from HBase (Appendix C), and six from Kafka (Appendix D). We detailed these numbers in Table 11.

5.2. Adjustment categories

Throughout our investigation, we identified five distinct categories of adjustments:

- **Fundamental principles adjustments:** These adjustments are based on the fundamental principles of log severity levels. They happen when there is a disconnect between the choice of a log's severity level and the fundamental principles that govern those levels. For example, a log statement may be mistakenly marked as critical *e.g., Error* or *Fatal* level, when, in reality, it is an informative situation (*Info* level). This type of adjustment is necessary to align log statements with the appropriate severity of the events they represent.
- **Historical practice adjustments:** Adjustments in this category come from an analysis of the code history and project issues. For instance, if certain log severity levels were adjusted in specific scenarios in the past, similar adjustments might be applied in current situations to maintain coherence and adherence to the project's historical practices.
- **Experience-driven adjustments:** Unlike the previous categories, these adjustments arise from the practical experience of developers and system operators in troubleshooting and diagnosing faults. They reflect a deep understanding of actual log needs when specific information may become more critical over time or in certain contexts. For example, a log initially considered low severity level (*Trace* or *Debug* level) may be elevated to a higher level (*Info* or *Warn* level) based on experience, knowing

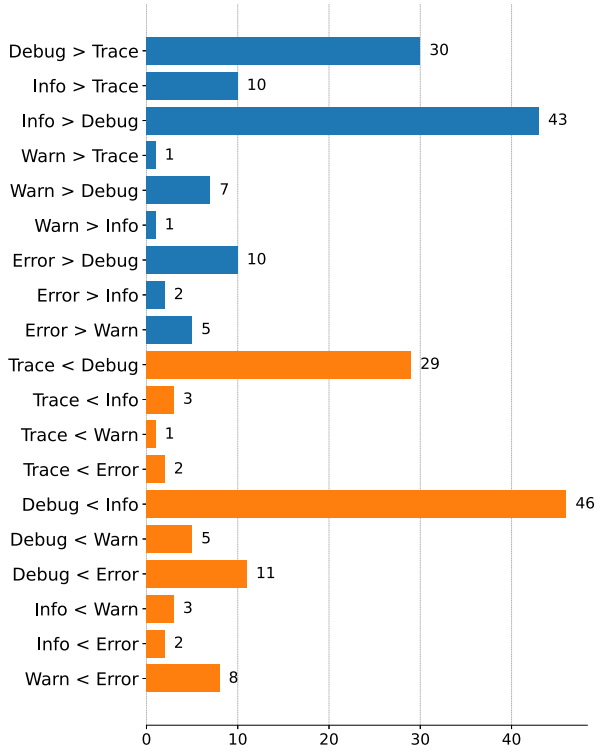

Fig. 19. Kafka: Adjustment occurrences.

that this information is often valuable for troubleshooting specific problems.

- **Guideline-based adjustments:** Here, the project prescribes adjustments by official documentation, which provides directives on logging practices.
- **Logging library swap adjustments:** Adjustments caused by switching logging libraries with different severity level sets.

Fig. 20 presents an overview of the adjustment categories by each selected system.

5.3. Fundamental principles adjustments

This category covers adjustments related to a lack of respect for the application fundamentals of each severity level. It also includes cases that show a lack of comprehension of how critical an event or information is in system behavior analysis activities during the choice of the initial severity level.

5.3.1. Misconception of log severity levels

In the selected systems, we frequently encountered evidence of errors in choosing log severity levels. This evidence is present in discussions explaining why the log statement does not align with the initially assigned severity level.

For example, we have found cases where a severity level adjustment for *Info* happens deliberately exclusively to facilitate debugging on YARN-1839:

Debug < Info[13]

"Changed some NMToken related debug level to info so as to make debug easier".

In another issue (HDFS-14759), the severity level of a log statement with only state values is reset to *Debug* after being changed to *Info* without previous discussion, indicating the inadequacy of the first severity level adjustment.

We also found very specific values registered at the *Info* level, which recall a type of over detailing (MAPREDUCE-5766, YARN-1022), and even values that do not seem to be missing from any software process, as described in issue YARN-10369:

Info > Debug

"We changed this to DEBUG internally years ago and haven't missed it".

Some of the discussions revealed intentionality to maintain certain severity levels for analysis or debugging purposes despite them being inappropriate for the situation, as present in the following statement from an attenuation adjustment on MAPREDUCE-5766:

Info > Debug

"I understand the desire to move ping requests and JVM asking for task messages to debug, but do we really want to move status updates to debug? They are particularly useful for debugging . . . "[14]

All these issues illustrate developers' dilemma in categorizing log severity: balancing the need for detailed information during debugging with the risk of cluttering the log with too many details.

But this misconception about LSL does not just happen between the *Debug* and *Info* levels. It also occurs when developers classify benign messages as troubles (HDFS-6998, MAPREDUCE-4570), error events that the client can handle or do not require immediate action, are classified as an *Error*, but rather a *Warn* (HADOOP-10274, HDFS-14760).

Error > Warn

". . . at this point, our hdfs installation wants to make sure no 'ERROR' is logged if it's not really an error that should/can be actionized". (HDFS-14760)

5.3.2. Valuable information

We identified adjustments when valuable troubleshooting information was hidden from the production environment log due to the use of a *Debug* level, as shown in (HADOOP-12789):

Debug < Info

"Knowing exactly what classpath ApplicationClassLoader has is a critical piece of information for troubleshooting. We should log it at the INFO level".

This sentence shows a lack of understanding of system behavior and the importance of information when the severity level was initially chosen. In the same context of troubleshooting, events causing execution errors did not leave clear clues about the problem that occurred, which led to an escalation of *Warn* to *Error* to facilitate the understanding of unexpected behaviors:

Warn < Error

"The socket is closed silently and nothing and an non understandable exception will be thrown". (HADOOP-1034)

[13] Each citation in this section is preceded by the type of severity level adjustment it discusses.

[14] Please note that the citations from Jira in this article are faithful reproductions of the original texts. As such, they may contain grammatical errors and awkward language constructions that are inherent in the source material.

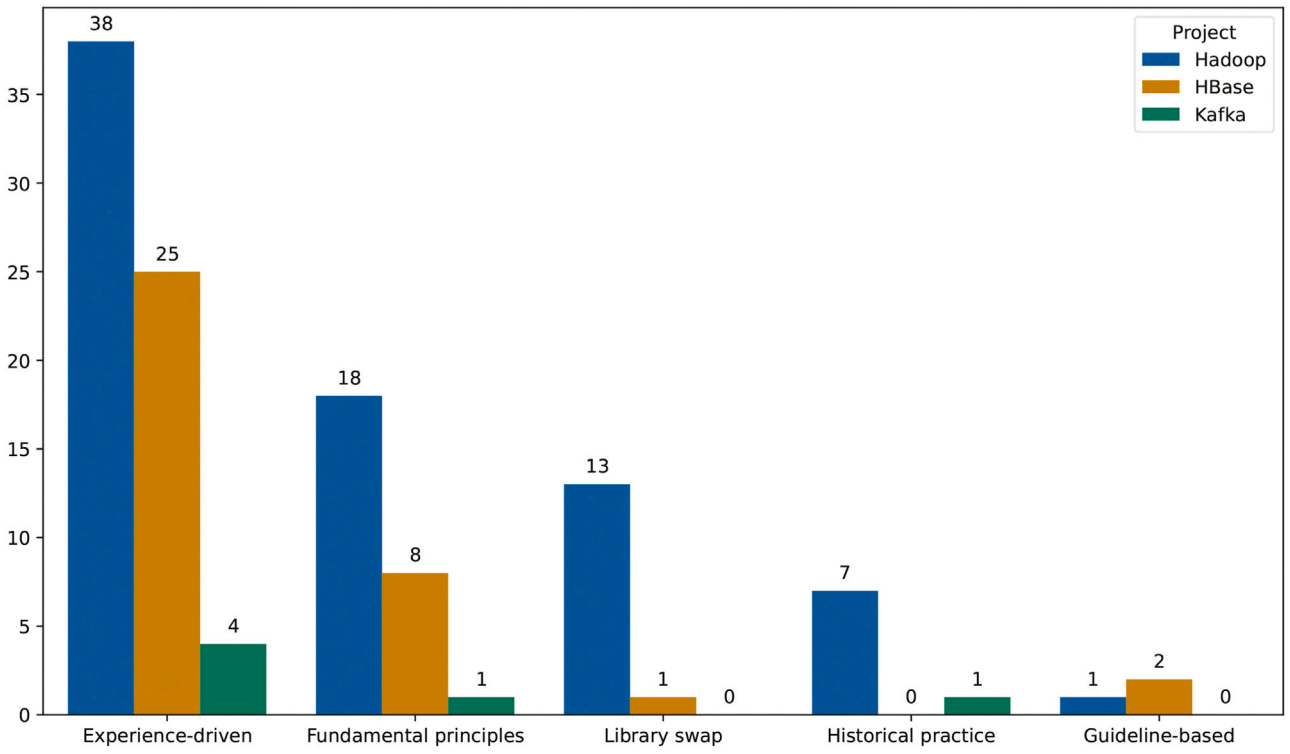

Fig. 20. Adjustment categories by system.

Identifying which information is more critical than others is important to avoid unintended consequences in log data. Such is the case with HADOOP-14987, an issue that was opened to enhance logging for troubleshooting purposes. In this issue, the developers reach a consensus on adjusting a single log statement from *Debug* to *Info* within a cluster of *Debug* statements. At first glance, the log statement could be mistaken for another *Debug* level entry due to its surrounding context. However, this targeted adjustment adds significant information to the production log without creating excess verbosity.

5.4. Historical practice adjustments

Nowadays, Incident Management Systems (IMS) like Jira play an essential role in creating a knowledge base by compiling code changes during the development and evolution of software systems. This database becomes a valuable tool for developers, not only for consultations when implementing new changes but also for supporting discussions with historical practices.

This principle also applies to adjusting LSL. Generally, discussions in this context are briefer, as they often reflect previously debated issues. Below, we highlight excerpts from some issues that exemplify this *modus operandi*.

In the two first situations, we find a direct comparison of the same situation with different severity levels:

Debug < Error

"... and each of them contains a log statement, most of them set the logging level to ERROR. However, when catches RuntimeErrorException in line 348 (r1839798), the logging level of log statement in this catch clause is DEBUG ..." (HADOOP-15942)

Warn < Error

"Therefore, I think these 3 log statements should be assigned ERROR level to keep consistent with other code snippets". (YARN-9349)

In the next one, rather than analyzing the context of the message and the log statement, the developer prefers to rely on the frequency of one severity level compared to another in the same situation:

Warn < Error

"After had a look on whole project, I found that there are 8 similar code snippets assign the ERROR level, when doTransition() occurs InvalidStateTransitionException. And there are just 3 places choose the WARN level when in same situations. Therefore, I think these 3 log statements should be assigned ERROR level to keep consistent with other code snippets". (YARN-9349)

Sometimes, there are situations where the assigned level of an event is clearly wrong, such as an *Error* level given to an event that does not affect the system's normal functioning. In these cases, the decision on how to handle the event can be based more on existing system practices rather than on logical reasoning:

Error > Info

"I think the logging practices should be consistent in similar contexts ... Checking more code, indeed there are inconsistency in those codes write() with info log level, create with error log info, while others with warn and info log level". (HDFS-14340)

5.5. Experience-driven adjustments

This category includes adjustments motivated by observing log data generated by software systems. Among them are some of the most apparent reasons to adjust LSL, as well as extraordinary exceptions that seem to deny the fundamental principles of the log.

5.5.1. Excessive logging

The leading cause of LSL adjustments is the production of excessive log entries, which we refer to as *"verbose logging"*. We found issues indicating verbosity for the *Error*, *Warn*, *Info*, and *Debug* levels. A specific characteristic of this situation is that the adjustments are always attenuated toward the *Debug* level, except when the source level is *Debug*, which attenuates toward the *Trace* level.

Except for *Debug* > *Trace* attenuation, the motivation for adjustments of this type aims to avoid overloading the production of log data, which impacts system performance and requires strategies to persist challenging amounts of data. The excessive amount of logs produced is explicitly mentioned in the issues excerpts below:

Info > *Debug*

"... we get this log printed about 50,000 times per second per thread" (KAFKA-13037)

Info > *Debug*

"I'm seeing 8gb resourcemanager out files and big log files, seeing lots of repeated logs (eg every rpc call or event) looks like we're being too verbose in a couple of places". (MAPREDUCE-3692)

Warn > *Debug*

"Each input record will contribute to one line of such log, leading to most of the tasks' syslog > 1GB". (HADOOP-11085)

Warn > *Debug*

"We saw an instance where this log message was generated millions of times in a short time interval, slowing the NameNode to a crawl. It can probably be changed to log at DEBUG level instead". (HDFS-11054)

Error > *Trace*

"... at the end of the test 1M rows always have arrived at the target cluster". (HBASE-12419)

The last quotes present different metrics to record excessive log production in production:*"50000 times per second"*, output files between 1 GB and 8 GB of data, *"generated millions of times"*. This excessive generation can impact the tasks that consume this data in production.

In this same context, we found issues that aim to solve the problem of excessive logging in production through an *Info* > *Debug* setting. However, there is resistance from developers who argue that if the adjustment remains, important information for debugging will be lost, suggesting that debugging is being done in a production environment (MAPREDUCE-3692), as described previously.

However, as previously stated, we also encounter situations in which the *Debug* log statement was explicitly labeled as "verbose" in the issue discussions (HADOOP-10015, HADOOP-18574, HBASE-9371, HBASE-16220, HBASE-17540), making the data so "obscure" that it makes debugging difficult:

Debug > *Trace*

"... logs too much, so much so it obscures all else that is going on". (HBASE-7214)

5.5.2. Exceptional adjustments

We found evidence that adjusting log levels is a strategic decision that varies with the system's development stage. We observed instances where messages typically associated with the *Debug* level were temporarily escalated to a higher severity level. This approach differs from the *"crutch"* method in debugging, discussed earlier under *Fundamental Principles Adjustments*.

For example, during the initial stages of system development, certain information might be really important for monitoring and is thus logged at a more severe level, like *Info* or *Error*, to ensure visibility. As the system stabilizes, these logs can generate excessive data or become less critical, warranting a reversion to the *Debug* level.

This practice is evident in MAPREDUCE-4614, where developers decided to keep a *Debug* statement at the *Info* level temporarily for system stabilization:

Debug < *Info*

"... maybe we can have them at INFO level while stabilizing the system, and change them to DEBUG at a later point. Otherwise, DEBUG level + a change in the client log level should work ... I'm ok with INFO during stabilization". (MAPREDUCE-4614)

We found a similar situation in (YARN-2704), where a *Debug* log was initially proposed to be escalated to *Info* but ultimately adjusted to *Error*. The motivation behind this decision reflects an understanding of the log's frequency and the importance of its visibility for effective analysis:

Debug < *Error*

"Regarding the info/debug level logs, these logs are all low frequency logs, by default it's only 1 day a time(renew interval). And it's so much easier to debug in info level than debug level. maybe in info level while stabilizing this feature?" (YARN-2704)

In this case, the decision to aggravate the log severity was driven by the recognition that rare yet important logs could be overlooked if left at the *Debug* level amidst a large volume of log entries.

5.5.3. Understanding the functioning and usage of the system

The user's experience utilizing a system's functionalities also contributes to the log adjustment process. For example, understanding which steps must be followed for a system process to be successful:

Info < *Warn*

"If users don't create all topics before starting a streams application, they could get unexpected results ... Also, this PR changes the log level from 'INFO' to 'WARN' when metadata does not have partitions for a given topic" (KAFKA-6802)

In this case, the log severity adjustment will help identify the reason for possible unexpected behavior in the application stream due to mandatory resources not created by the user.

Another example of experience with the system is issue HDFS-14760 initially requesting an *Error* > *Info* attenuation. However, a developer clarifies the understanding of the event's consequences, and the log statement goes to the *Warn* level:

Error > *Warn*

"I think at least WARN level is warranted, given that this is an issue which could potentially cause directories to exceed their quota or be blocked even though their quota is not yet met". (HDFS-14760)

5.5.4. Exceptions from legacy code in evolving systems

As systems evolve, new software components often must communicate with pre-existing legacy components. This evolution, while necessary, can sometimes lead to communication failures between new and old interfaces, potentially causing disruptions in the entire process.

A notable instance of this challenge is documented in the issue KAFKA-5704. Kafka is a distributed streaming platform; in this case, older versions of Kafka brokers could not handle new requests for checking and creating missing topics, resulting in an *UnsupportedVersionException* that halted the entire process. The solution proposed in the issue reflects a nuanced understanding of both software evolution and effective logging practices:

Table 13
Preliminary observations.

#	Text	Issue
[PO1]	There are instances of Debug to Info adjustments exclusively to facilitate the debugging process.	MAPREDUCE-5766, HDFS-14759, YARN-1839, MAPREDUCE-3692
[PO2]	Log statements consisting solely of state values are frequently associated with debugging.	HDFS-14759
[PO3]	Logs that excessively detail events and states were generally classified as debug logs, indicating a possible preference for finer granularity at this severity level.	MAPREDUCE-5766, YARN-1022
[PO4]	Assigning client-handled errors to levels intended for reporting system failures may not be appropriate, as these errors might be more suitably logged at a less severe level.	HADOOP-10274,HDFS-14760
[PO5]	Vital information should avoid being logged at the Debugging Purpose severity levels to ensure that essential events are not obscured and remain readily identifiable.	HADOOP-1034, HADOOP-12789, HADOOP-14987, KAFKA-6802
[PO6]	The significance and rarity of logged information should be directly proportional to the severity level chosen; more critical and uncommon events warrant higher severity levels.	HADOOP-12789, YARN-2704
[PO7]	Log entries that occur frequently are best suited for *Debugging Purpose* severity levels, facilitating a focused analysis without overwhelming the log with less critical information.	KAFKA-13037, MAPREDUCE-3692, HADOOP-11085, HDFS-11054, HBASE-12419, HBASE-7214, HBASE-15954
[PO8]	The severity level chosen for a log statement can change as the system matures, reflecting the change in the importance and frequency of logged events.	MAPREDUCE-4614,HDFS-14760, YARN-2704
[PO9]	Normal and benign behaviors are not compatible with the levels of Warning and Failure Purposes.	HDFS-6998, MAPREDUCE-4570

Error > Debug

"We should probably just catch it, log a message, and allow things to proceed … This change handles the UnsupportedVersionException by logging a debug message and doing nothing" KAFKA-5704

This approach entails catching the exception and logging a *Debug* message, a strategy that balances the need for error tracking with the necessity of maintaining system functionality. By choosing a *Debug* level log, the developers demonstrated an informed decision to prioritize system continuity while still keeping a record of the issue for future diagnostics. This decision illustrates a practical resolution to the challenges posed by legacy systems in modern software development, ensuring system robustness and user continuity despite potential incompatibilities.

5.6. Logging library swap adjustments

Log libraries have different severity levels. Therefore, when we switch from one logging library to another, there is no guarantee that the current severity levels will be maintained. This event happened, for example, in Hadoop, where at least two logging library substitutions occurred: first from Java Util Logging (seven severity levels) to Apache Commons Logging (six severity levels) and then from Apache Commons Logging to SLF4J (five severity levels). To map the levels between the different libraries, we used the concepts of equivalence between severity levels presented in our previous study (Mendes and Petrillo, 2021). These circumstances came up with the following adjustments:

- Equivalence adjustments: *Fine* ≡ *Debug*, *Finer* ≡ *Debug*, *Finest* ≡ *Trace*, *Warning* ≡ *Warn*, *Severe* ≡ *Fatal*. These updates refer to the swap from Java Util Logging to Apache Commons Logging. Each update indicates an equivalence of severity levels between the two libraries.
- Attenuation adjustment: *Fatal* > *Error*. The second highest update occurrence in Hadoop, *Fatal* > *Error* (65), is caused by swapping the code to the SLF4J library in the log statements, which has only five severity levels, not including *Fatal* — its most severe level is *Error*; the library previously used was Apache Commons Logging.

5.7. Guideline-based adjustments

A guideline can be a written document, or it can also be one of the initial issues of the project that seeks to establish the standard for how to log in.

A logging guideline was proposed in version *0.3.0* of Hadoop through issue HADOOP-211 endorsing a comprehensive and unified approach to logging. The key aspects of this guideline focus on establishing a consistent logging format across all Hadoop subsystems. This proposal aims to enhance log analysis and debugging by ensuring that logs are structured and emphasizes the importance of clear annotation in log entries, including timestamps, logging levels, and traceability to the originating subsystem. This initiative represents a significant step in standardizing logging practices within Hadoop.

There are also references to conventions[15] not respected in HBase issues, as in this adjustment to reduce the amount of logs generated by default:

Debug > Trace

"We should bring REST server to be on par with the RS (RegionServer) level log conventions. Individual requests to be only logged at the TRACE level". (HBASE-15954)

5.8. Preliminary observations and potential heuristics

Throughout the analysis of adjustments in Phase 2, we found insights that suggest patterns in how severity levels are assigned and adjusted during software development and operations. We group these insights into Table 13 and refer to them as *"preliminary observations"* (PO).

Complementary to the insights, we also found some textual fragments in the issues with a more imperative nature, containing direct recommendations regarding adjusting the LSL. We refer to these texts, grouped in Table 14, as *"potential heuristics"* (PH) because they suggest direct applicability.

[15] We were unable to find a record of these conventions.

Table 14
Potential heuristics (PH).

#	Adjustment	Potential raw heuristic	Issue
[PH1]	*Warn<Fatal*	*"ReplicationPeer logs at WARN level aborting server instead of at FATAL"*	HBASE-7037
[PH2]	*Warn<Error*	*"Only IOException is catched and logged (in warn). Every Throwable should be logged in error. Eg: a RuntimeException occurs in the writeBlock() method. The exception will not be logged, but simply ignored. The socket is closed silently and nothing and an non understandable exception will be thrown in the DFSClient sending the block...."*	HADOOP-1034
[PH3]	*Info<Warn*	*"I'd up the info on JettyBugMonitor line 80 a warning since it's logged once and perhaps disabling a feature the user thinks they have. [...] I changed the "info" to "warn" as you suggested on commit."*	MAPREDUCE-3184
[PH4]	*Debug<Info*	*"This message should be at INFO. it doesn't happen often and knowing what plugins decide about file deletions is important all the time."*	HBASE-23250
[PH5]	*Error>Warn*	*"so the error may not really be an error if client code can handle it"*	HADOOP-10274
[PH6]	*Error>Info*	*"There's a warn log before that exception. The code still work well even with exception thrown out. [...] yes, the code still works well in all of three situations [...] Thus, it seems that the first log statement should be assigned to a lower level, INFO "*	HDFS-14340
[PH7]	*Error>Warn, Error>Info*	*"ClassFinder logs error messages that are not actionable, so they just cause distraction [...] Simple patch that changes lines to warn and info depending on how likely it is to be actually worth looking at."*	HBASE-9120
[PH8]	*Warn>Debug*	*"We will need to change the log level to debug to avoid such excessive logging."*	HADOOP-11085
[PH9]	*Warn>Debug*	*"There are many exceptions that the client can automatically handle [...] Attaching a patch that changes the warning log to a debug log [...]"*	HADOOP-13552
[PH10]	*Warn>Debug*	*"If this is really harmless, why do we log? [...] Just change it to debug? Ok"*	HBASE-25642
[PH11]	*Info>Debug*	*"DEBUG level is good for frequently logged messages, but the shutdown message is logged once and should be INFO level the same as the startup"*	HDFS-10377
[PH12]	*Info>Debug*	*"That's why we have the debug log level right? Devs can toggle it if you want to see all AM transitions but doesn't inflict huge logs on normal users. "*	MAPREDUCE-3692
[PH13]	*Info>Debug*	*"we get this log printed about 50,000 times per second per thread [...] I completely agree that logging this at INFO on every iteration is wildly inappropriate, I just didn't push it at the time since I figured someone would file a ticket if it was really bothering people. And here we are"*	KAFKA-13037
[PH14]	*Debug>Trace*	*"CleanerChore logs too much, so much so it obscures all else that is going on. [...] Dropped the most prominent logs to trace, so if you really need the info, you can get at it (helpful for debugging), but shouldn't come up in most cases."*	HBASE-7214
[PH15]	*Debug>Trace*	*"Messages like this can significantly accumulate in regionserver logs when a cluster is carrying empty tables. [...] Because many deployments run with DEBUG as the default log level, especially when trying to track down other production issues, this message is better logged at TRACE level."*	HBASE-16220
[PH16]	*Debug>Trace*	*"We should bring REST server to be on par with the RS level log conventions. Individual requests to be only logged at the TRACE level.[...]"*	HBASE-15954

Each table entry (Tables 13 and 14) represents filtered content from discussions found in the selected Jira issues.[16] Although insights are not yet definitive heuristics, they form the foundations that will be iteratively examined and refined in Phase 3 to formalize them.

6. Prescriptive phase (phase 3) | results

In this section, we present the results of the Prescriptive Phase of our study, where we transform insights and patterns identified in the Fundamental Principles of Log Severity Levels (Section 2.1) and the Explanatory Phase (Phase 2) into practical heuristics for LSL decisions. We synthesize a set of heuristics that reflect the complexities and nuances of log severity adjustments, providing a structured path for making informed decisions in log severity classification.

The relationship between the final heuristics, potential heuristics, preliminary observations, and the studied issues is detailed in Table 15, as discussed in Section 3.4.1.

6.1. Fundamental principles based heuristics

From the fundamentals principles, we classify the severity levels into two categories, first about the environment and then about their purpose according to Fig. 3. We generate specific heuristics for each category that help classify log statements within that category. However, considering that the set of log severity levels depends on the used library,[17] we present heuristics aimed at the purpose of the level (*Debugging, Informational, Warning,* and *Failure*) rather than the level itself.

6.1.1. Environment heuristics

The initial heuristic separates the log statements into two groups: sentences targeting the system *development environment* (*Debugging purpose*) and sentences targeting the *production environment* (*Informational, Warning and Failure purposes*). The log statements of each group have different target audiences, and this, in addition to implying the choice of the LSL, also impacts the construction of their messages. Therefore, the construction of log messages is a key factor in achieving accurate severity classification.

For example, a message made up only of state or variable values can be effective in helping to solve specific development environment problems; these targeted details support developers in identifying and addressing issues during the development process. However, for system managers, in a production environment, who need to analyze the behavior of systems without having developed them, the messages must be written clearly to help them troubleshoot tasks. These messages must be written to provide a quick and accurate interpretation of the recorded events, helping to identify and effectively resolve problems.

[16] The citations from Jira in this article are faithful reproductions of the original texts.

[17] For example, while the most severe level in one logging library is the *Fatal* level, it may be the *Error* level in another.

Table 15
The relationship between heuristics, potential heuristics (PH), preliminary observations (PO), and Jira issues. The "Related Issue" column presents the issues that support the heuristic. The "Freq(HD/HB/K)" column shows the frequencies of issues related to the Hadoop ("HP"), HBase ("HB"), and Kafka ("K") projects.

Heuristic	Related PHs	Related POs	Related issues	Freq HP/HB/K
[H1]	–	–	–	–
[H2]	–	–	–	–
[H3]	–		HDFS-14759, MAPREDUCE-5766, YARN-1022	3/0/0
[H4]	–	–	–	–
[H5]	–	–	HADOOP-10274, HADOOP-1034, HADOOP-10657, HADOOP-96, HDFS-14238, HDFS-14395, HDFS-14521, HDFS-14759, HDFS-14760, HDFS-6998, MAPREDUCE-3348, MAPREDUCE-4570, MAPREDUCE-5766, MAPREDUCE-7063, YARN-1022, YARN-10369, YARN-1608, YARN-1839, HBASE-20665, HBASE-7037, HBASE-9120, HBASE-23047, HBASE-14042, HBASE-8940, HBASE-10906, HBASE-20447, KAFKA-9540	18/8/1
[H6]	–	[PO1]	HDFS-11593, HDFS-14759, MAPREDUCE-3692, MAPREDUCE-5766, MAPREDUCE-7063, YARN-1608, YARN-1839, HBASE-20554, HBASE-20665	7/2/0
[H7]	–	[PO2]	HDFS-14759, MAPREDUCE-5766, YARN-1022	3/0/0
[H8]	–	[PO4]	HADOOP-10274, HADOOP-13552, HADOOP-17597, HDFS-14760	4/0/0
[H9]	–	[PO5]	HADOOP-1034, HDFS-11593, HADOOP-12789, HBASE-13675, HADOOP-14987	5/0/0
[H10]	–	[PO8]	HDFS-14760, MAPREDUCE-4614, YARN-2704, KAFKA-5704	3/0/1
[H11]	[PH1], [PH2]	–	HADOOP-1034, HBASE-7037	1/1/0
[H12]	[PH5]	[PO4]	HADOOP-10274, HDFS-14760, HBASE-10906, HBASE-14042, HBASE-16589, HBASE-27391, KAFKA-9540	2/4/1
[H13]	[PH6]	–	HADOOP-17597, HDFS-6998, HBASE-27391	2/1/0
[H14]	[PH7]		HADOOP-17597, HDFS-6998	2/0/0
[H15]	[PH7]	[PO4]	HADOOP-10274, HADOOP-17597, HDFS-14760, HBASE-9120, HBASE-26189, HBASE-27391	3/3/0
[H16]	[PH3]	–	HDFS-14760, KAFKA-5704, KAFKA-6802	1/0/2
[H17]	[PH4], [PH11]	[PO6]	HADOOP-10657, HADOOP-12789, YARN-2704, HBASE-23250	3/1/0
[H18]	[PH4], [PH11]	[PO6]	HADOOP-10657, HADOOP-12789, MAPREDUCE-3184, YARN-2704, HBASE-5582, HBASE-7037, HBASE-8940, HBASE-23250, KAFKA-6802	4/4/1
[H19]	[PH11]	–	HDFS-10377	1/0/0
[H20]	[PH8], [PH11]	[PO7]	HADOOP-7858, HADOOP-8075, HADOOP-8932, HADOOP-9135, HADOOP-9582, HADOOP-10015, HADOOP-11085, HADOOP-15441, HADOOP-17836, HADOOP-18065, HADOOP-18574, HDFS-8659, HDFS-9906, HDFS-10377, HDFS-13692, HDFS-15197, MAPREDUCE-3265, MAPREDUCE-3692, MAPREDUCE-3748, YARN-1892, YARN-2213, YARN-3350, YARN-4115, YARN-5693, YARN-7727, YARN-8459, YARN-10997, HBASE-12419, HBASE-12539, HBASE-12461, HBASE-15582, HBASE-15954, HBASE-24524, HBASE-25483, HBASE-25556, HBASE-25642, HBASE-26443, HBASE-27588, KAFKA-4829, KAFKA-13037, KAFKA-13669	27/11/3
[H21]	[PH8], [PH13]	[PO2], [PO3]	HDFS-11593, HDFS-14759, MAPREDUCE-5766, YARN-1022	4/0/0
[H22]	[PH8], [PH12]	[PO7]	HADOOP-7858, HADOOP-10015, HADOOP-10343, HADOOP-11085, HADOOP-16708, HDFS-11054, MAPREDUCE-3692, HBASE-12419, HBASE-15954, HBASE-20554, KAFKA-13037	7/3/1
[H23]	[PH12], [PH15]	[PO2], [PO3], [PO7]	HADOOP-11085, HADOOP-18574, HDFS-11054, MAPREDUCE-3692, HDFS-14759, MAPREDUCE-5766, YARN-1022, HBASE-7214, HBASE-9371, HBASE-12419, HBASE-12461, HBASE-15954, HBASE-16220, HBASE-20701, HBASE-20770, HBASE-23687, HBASE-27079, KAFKA-13037	7/9/1
[H24]	[PH15], [PH16]	[PO3], [PO7]	HADOOP-10015, HADOOP-11085, HADOOP-18574, HDFS-11054, MAPREDUCE-3692, MAPREDUCE-5766, YARN-1022, HBASE-7214, HBASE-9371, HBASE-12419, HBASE-12461, HBASE-15954, HBASE-16220, HBASE-20701, HBASE-20770, HBASE-23687, HBASE-27079, KAFKA-13037	7/10/1

Heuristic #1: *The Trace and Debug levels MUST classify log statements targeting exclusively the development process.*

Heuristic #2: *The Info, Warn, Error, and Fatal levels MUST classify log statements targeting the production environment.*

Heuristic #3: *Log statements intended for the development environment MAY focus on specific aspects of system states, such as variable values, without requiring the broader context generally necessary for a wider understanding.*

Heuristic #4: *Log statements intended for the production environment MUST have messages written in well-structured and clear sentences, aiming to facilitate human understanding.*

Next, we move on to the second category of heuristics, relating to the purpose of log severity levels.

6.1.2. Purposes' heuristic

The four logging purposes (*Debugging*, *Informational*, *Warning*, and *Failure*) encompass intent for all severity levels in the log libraries and suggest the main reason we log framing log statements objectively. Those purposes generated the following heuristic.

Heuristic #5: *The intent behind logging MUST be analyzed when selecting severity levels, ensuring they are chosen according to their intended purposes (Debugging, Informational, Warning, and Failure). The purposes SHOULD guide the selection of severity levels to maintain clarity and usefulness of log data.*

6.2. Heuristics from the phase 2 | heuristics from the insights

The investigation into the causes of LSL adjustment also contributed to heuristics, namely:

> ***Heuristic #6:*** *Debug levels MUST NOT be masked as Info, as this may generate excessive logs.*

> ***Heuristic #7:*** *Log statements composed solely of state values SHOULD be classified with Debugging Purpose levels.*

> ***Heuristic #8:*** *Errors that the client can handle MUST NOT be classified with Failure Purpose severity levels.*

> ***Heuristic #9:*** *Important information SHOULD NOT be classified with Debugging Purpose severity levels, as this may make it difficult to recognize the occurrence of the event.*

> ***Heuristic #10:*** *Log statements strategically adjusted in previous versions SHOULD be periodically reviewed to ensure they remain relevant and do not contribute to log overflow. The appropriate severity level for a log statement MAY change as the system evolves.*

6.3. Heuristics from the phase 2 | potential heuristics

In this section, we propose heuristics based on the issue excerpts presented in Table 14.

6.3.1. Heuristics related to exceptions

By its very nature, there is a tendency to link exceptions to the severity levels of the *Failure* purpose, as *Fatal* and *Error*. However, it is important to consider the context and consequences of exceptions, as many cases do not result in critical situations for the system. In other cases, they do.

PH1 and PH2 are examples of exceptions causing software interruptions, such as server aborts (described in PH1) or socket closures (described in PH2), which should interrupt significant software operations.

Despite this, these log statements had a less critical severity level, and therefore, both underwent a severity aggravation, from a *Warning* level to a *Failure* level. Furthermore, both are involved in throwing Throwable[18] class exceptions, described in other Jira issues as events aligned with the levels of the *Failure Purpose*, adding concrete examples of application.

> ***Heuristic #11:*** *Log statements indicating server shutdown or failure, or socket closures, MUST be aligned with the Failure Purpose levels.*

Following, we will enumerate some of the cases in which exceptions are not so critical from the point of view of LSL:

- An exception was thrown, but the user can deal with the generated situation himself [PH5];
- An exception was thrown, but the 'code' continues to work well [PH6];
- An exception was thrown, but the situation caused by it does not require immediate action for the system to remain functional [PH7].

From the situations listed above, we can infer new heuristics:

> ***Heuristic #12:*** *Severity MUST be classified with Failure Purpose levels only if it is a real issue that impacts code functionality and cannot be handled by the user or client.*

> ***Heuristic #13:*** *If an exception is thrown, but the system continues to operate normally without impacting its functionality, then the log statement SHOULD NOT be classified with a failure-finality level.*

> ***Heuristic #14:*** *If a log statement reports an issue but the system does not stop, it SHOULD be classified with a Warning Purpose level rather than a Failure Purpose level.*

> ***Heuristic #15:*** *In exception events that do not require immediate action, this event SHOULD NOT be classified at a Failure Purpose level. Less severe log severity levels MUST be considered.*

6.3.2. Disabling features

PH3 reports a case about an event that turns off a feature that users may think they will have access to. We can consider this situation an unexpected event, as described in *Warning Purpose* definition. However, the event was initially logged with severity level *Info* despite being described as low frequency and may go unnoticed.

The adjustment in this case was an aggravation for *Warning*, and from it we can suggest the following heuristics:

> ***Heuristic #16:*** *If an event can disable a functionality expected by users, this event MUST be classified with a Warning Purpose severity level.*

6.3.3. Uncommon events

Continuing with the idea that important information can go unnoticed depending on the severity level used, PH4 describes a situation regarding deleting files, classifying them as "important all the time", but which is at the *Debug* level.

PH11 describes a similar situation concerning shutdown logs, which are recorded only once and would have a severity corresponding to a startup message. As described before, statements that describe the starts and ends of expected events by the system are suitable for the *Informational Purpose* severity level.

> ***Heuristic #17:*** *Uncommon events SHOULD NOT be classified with Debugging Purpose levels. More severe log levels SHOULD be considered.*

> ***Heuristic #18:*** *Log statements reporting rare but significant events SHOULD be classified with Informational and Warning Purpose levels.*

> ***Heuristic #19:*** *Expected startup and shutdown messages SHOULD be classified with Informational Purpose severity levels.*

6.3.4. Frequent and detailed logs

Among the potential heuristics in Table 14, we found repeated indications that frequent messages are suitable for *Debugging Purpose* severity levels [PH8, PH9, PH10, PH11, PH12, PH13]. At the same time, they are entirely inappropriate for the *Informational Purpose* level [PH13], as they can overload users with messages that do not have criticality [PH12], in addition to having the capacity to make it challenging to understand logs destined for the production environment [PH14]. As described in PH13, frequent messages can be associated

[18] "The Throwable class is the superclass of all errors and exceptions in the Java language" (Oracle, 2018)

with log statements in iteration regions to detail the events and states that occurred, which explains their high incidence.

Heuristic #20: *Debugging Purpose levels MUST classify messages that are logged frequently.*

Heuristic #21: *Summary logs and detailed logs SHOULD correspond to Debugging Purpose levels.*

Heuristic #22: *Users MUST NOT be overwhelmed by excessive logs under normal circumstances unless it is critical for the user to be aware of the event.*

In this context, we must also consider the severity level chosen for producing logs in a production environment. The default is the *Info* severity level. However, in PH15, we see deployments with the *Debug* level by default. This choice leads us to consider the severity of the more detailed events and states that should be logged, whether at the *Debug* or *Trace* level.

Heuristic #23: *When the standard for log generation in a production environment is the Info level, the Debug level SHOULD be preferred for detailed information. If the standard is the Debug level, the Trace level SHOULD be considered to avoid generating excessive Debug logs.*

We should also consider how to choose between *Debug* and *Trace* levels on frequent log occasions, as even the same debug process can become obscure when faced with excessive logs [PH15]. In these situations, we can resort to the following heuristic:

Heuristic #24: *In cases of excessive logs that make the debugging process unclear, the severity level of the most prominent log statements SHOULD be downgraded to the Trace severity level.*

7. Discussion

In this section, we discuss our study's results and main findings of each of our three-phase methodology.

7.1. Descriptive phase (phase 1)

During the first phase, we presented quantitative data that provide an overview of LSL adjustments in the selected systems. Next, we will present the key findings of this phase.

7.1.1. The distribution of severity levels reinforces the idea of severity purposes

The percentage distribution of severity levels in the selected systems throughout the releases reinforces the concept of severity purposes presented in Section 2.1. The investigated systems' four most present severity levels, namely *Info, Debug, Warning, and Error* (in this order), go toward the proposed log purposes.

As the multiple severity levels found in the set of log libraries studied converge to the four most used severity levels in the investigated systems, be it *Info*, *Debug*, *Warning*, and *Error* (in this order), those go toward the proposed purposes: *Informative*, *Debugging*, *Warning*, and *Failure*, respectively.

7.1.2. There is a trend to take debugging-exclusive logs to the production environment

Info and *Debug* are the predominant severity levels in the log statements of the selected systems, as are the adjustments involving these two levels: *Info* > *Debug* and *Debug* < *Info*, in this order. In some cases, developers may choose to classify *Debug* messages as *Info* to capture more detailed information in production environments. However, this practice can lead to potential challenges, such as the generation of excessive logs that obscure critical events or add unnecessary load to the system. While this adjustment may be useful for debugging purposes, it highlights the need for more structured guidelines to balance verbosity and performance in production logging.

7.1.3. There is difficulty in distinguishing log statements between adjacent severity levels

1-degree severity adjustments, whether attenuating or aggravating, constitute the majority of the adjustments observed. This finding supports the idea that understanding adjacent severity levels is nuanced, and there may even be a biased understanding of them, complicating the choice of severity level.

As also noted by Li et al. (2017a), developers often make adjustments between adjacent severity levels in an effort to fine-tune log granularity without causing overload. However, this subtle distinction may lead to inconsistencies, especially in the absence of clear guidelines regarding the use of each level. The frequent adjustments between *Info* and *Debug*, often without a well-defined technical justification, suggest that individual interpretation plays a significant role. This trend highlights the need to clarify the roles of each severity level to reduce ambiguity and improve consistency in log management.

7.1.4. There is a tendency to assign excessively high severity levels to log statements initially

There is a prevalence of adjustments in the attenuation category. This prevalence suggests that developers tend to assign higher severity levels than necessary, which explains the need to reduce these levels over time. Additionally, this could indicate a tendency for some log statements to be perceived as less severe in subsequent releases or that, when writing new code, we need to observe its execution more carefully than when it becomes mature and potentially contains fewer bugs.

As observed by Li et al. (2020a), developers often perform reactive adjustments of log severity levels, especially during the final integration stages, to align them with production needs. However, when such adjustments are not part of a continuous review process, they may introduce technical debt by accumulating log statements whose severity levels are inadequate for their actual usage context.

7.2. Explanatory phase (phase 2)

In this phase, we focused on qualitative data, analyzing the text of the issues related to adjustments in the severity level of logs found during Phase 1 in order to find the reasons that guided such adjustments. Next, we present the main conclusions.

7.2.1. Dominance of experience-driven and fundamental logging principles adjustments

Phase 2 reveals that experience-driven adjustments constitute the bulk of the adjustments in all three projects. This trend suggests that practical, hands-on experience in troubleshooting and system operation significantly influences decisions related to LSL. As observed by Li et al. (2020a), developers often adopt improvised strategies for logging adjustments, proactively deciding where and when to log, or reactively adjusting severity levels during integration to meet production needs.

This intuitive and flexible approach is reflected in our study, where a large portion of observed adjustments respond to production demands and developer information needs. For instance, developers frequently adjust severity levels to highlight critical events or suppress excessive log output, thereby facilitating more efficient log management. While often improvised, such adjustments demonstrate a pragmatic use of logging to adapt to the specific conditions of each project.

Although adjustments guided by experience are more numerous, adjustments based on fundamental principles also have a substantial presence. These indicate a significant number of technically oriented adjustments aimed at aligning log statements with appropriate severity levels, supporting consistency and clarity in logging practices.

7.2.2. Lower incidence of guideline-based and historical practice adjustments

Adjustments based on guidelines and historical practices are the least prevalent. Interestingly, the former group is the rarest across all three projects. This fact could imply that formal guidelines and historical precedents play a relatively minor role in the decision-making processes for log adjustments compared to practical experience and fundamental logging principles.

However, these findings reveal a lack of guidelines for the project's logging practices. In this context, the recurrence of similar decisions across issues suggests that developers often rely on historical practices, documented implicitly in Jira issues, as a substitute for formal guidance.

These observations align with findings by Li et al. (2021a), who noted that duplicated log statements and code clones can lead to inconsistencies in severity levels. In our study, we similarly observed that, when facing similar events, developers tend to reuse severity levels from prior cases to promote consistency. While this pragmatic behavior aims at uniformity, it may also mask misclassifications and result in code smells, *e.g.,* logging non-critical events with high severity levels, contributing to log overload.

These results highlight the importance of having well-defined documentation or guidelines for severity level usage. Without them, the accumulation of adjustments based on precedent alone may introduce long-term maintenance issues and reduce the effectiveness of logging practices.

7.2.3. The prevalence of adjustments between info and debug levels

Phase 2 revealed that the reason for the adjustments between Debug and Info levels, mostly attenuations, is not guided by the fundamentals of logging. At the same time, they cannot be justified by experience either. Log statements at the *Debug* level tend to be aggravated to *Info* with the "excuse" that it would be easier debugging at the *Info* level. However, this practice is questionable, as such adjustments may lead to data overload and indicate a misinterpretation of the intended use of log severity levels.

7.2.4. Past strategic adjustments

Some logs were added in previous software releases to address challenges specific to that period. However, as software evolves, the relevance of these records may decrease, or their verbosity may become excessive. Periodically reviewing and adjusting these logs can help maintain the efficiency of the logging system.

7.2.5. Excessive production logs due to incorrectly classified debug -level statements

There are many adjustments made to address the excessive production of log entries in a production environment. In all cases found in our study, the solution was to attenuate the severity levels to those of the *Debugging Purpose*, which reveals an initial misclassification of severity level. These excessive logs can be helpful during development or debugging but can become an overhead in production environments.

This issue aligns with previous findings by Li et al. (2017b), who reported an attenuation from *Info* to *Debug* due to excessive logging noise. Our study reveals similar adjustments involving not only *Info* but also *Warn* and *Error* levels. In all such cases, severity levels were attenuated to reflect debugging purposes, suggesting that the initial classification was overly aggressive.

While excessive logs are useful during development, their presence in production can significantly impact performance and readability. These findings reinforce the need more rigorous strategies and clearer guidelines to manage severity levels effectively across environments.

7.2.6. More severe levels for silent problems

Silent issues can be challenging to diagnose due to log sparsity associated with more verbose severity levels, such as *Debug* or *Trace*. Identifying and properly handling these scenarios helps improve troubleshooting tasks. Using a more severe level can be crucial in these cases.

7.2.7. The log severity level is mutable

Phase 2 also reveals that in software development, adjusting log levels is a strategic decision that varies according to the development stage of the system. We observed cases where log statements carrying *Debug* level information, which are satellites to events associated with more severe levels, were temporarily escalated to a higher severity level. This approach differs from the "forced" adjustments from *Debug* to *Info* in debugging, discussed earlier in the previous subsection, by escalating the severity level of a log statement arbitrarily, which do not add value to log statements of other severity levels.

This mutability led us to understand that in a log statement, the severity level is not entirely dictated by its message. Rather, the severity level can vary as the system develops and evolves, even though the message itself remains the same. With each software version, the severity may attenuate or escalate depending on the circumstances in which the log statement is applied.

This observation is consistent with findings from Tang et al. (2022), who used Git history and a Degree of Interest (DOI) model to dynamically identify areas of code that require more attention. Their system automatically adjusts the severity levels of associated log statements, reflecting the developers' increasing interest in specific portions of the code as the importance of features evolves — even when the content of the log message remains unchanged.

7.3. Prescriptive phase (phase 3)

The Prescriptive Phase of our research highlights the challenges when choosing LSL. During our process of deriving heuristics from previous phases' insights, we encountered issues reflecting the dynamics of developers when creating and adjusting LSL.

7.3.1. Adapting log messages for target-environments

Our heuristic set begins by considering the importance of adapting log statements to the target environment, whether the development or production environment. During our investigation, it became clear that the relevance of a log entry is linked to the intended audience. Echoing the observations of Shang et al. (2015), who noted that "high-level logs are intended for system operators, and low-level logs are for developers and testers", we found that developers often need granular details for debugging, while system administrators require clear and concise logs to effectively troubleshoot problems. These two sides of the logs are translated by the ambient heuristics, which highlight the need to construct messages compatible with the chosen severity level.

7.3.2. Adaptation to the purpose of severity levels

Our proposal to group log severity levels by purpose - *Debug, Informational, Warning,* and *Failure* - provides a guiding tool for classifying log severity levels. This tool respects the diversity of sets of severity level sets across the different log libraries. Furthermore, it emphasizes that the concept of the log purposes is more important than specific level names, which are divergent across libraries. Our purpose-driven heuristics make it easier to select severity levels by focusing on a more comprehensive approach.

7.3.3. Dynamic nature of severity levels

Our results also highlight the importance of understanding the mutability to which severity levels are subject depending on the software development stage and the evolution of its functionalities. This observation aligns with the findings of Ding et al. (2015), who noted that log statements are often left in the code after their initial insertion, with severity level adjustments typically made during final code integration.

As a strategic measure, we observed that it is valid to aggravate severity levels temporarily to help troubleshoot unstable phases, and that certain log statements can become less severe from a strategic point of view as the system evolves.

7.3.4. Challenges of excessive logging

The most common severity adjustment context was excessive logging in production environments. This issue has been widely recognized in prior studies as a source of system overload and reduced log utility (Yuan et al., 2012a; Chen and Jiang, 2017a; Hassani et al., 2018; Zeng et al., 2019; Li et al., 2020a).

Sometimes, a temporary strategic measure can become an undesirable *modus operandi*, causing unwanted effects such as excessive log production. Based on this, we derive heuristics that recognize the fine line between temporary strategy and over-logging. Our heuristics advise *Debugging Purpose* levels for recurring details and more severe severity levels for events that, despite being rare, can be significant in elucidating problems.

7.3.5. The role of experience and knowledge

Experience has proven to play a fundamental role in discussions of log severity adjustments. It reflects the in-depth day-to-day knowledge of logging while also demonstrating absorption and constant reflection on the fundamental principles of logging. Sharing doubts and searching for practical solutions recorded, as recorded in system issues, were important for reflecting on and refining our heuristics.

7.3.6. Preliminary observations as heuristic foundations

The preliminary observations that emerged from our analysis of Phase 2 adjustments were instrumental in developing our final set of heuristics. These preliminary insights set the stage for exploration and synthesis in Phase 3.

7.3.7. Relation to automated classification approaches

While several recent studies have proposed automated methods for log severity classification using machine learning or large language models (*e.g.,* UniLog (Xu et al., 2024), DeepLV (Li et al., 2021b)), our approach serves a different purpose. Instead of automatically assigning severity levels, our heuristics aim to support developers' decision-making with interpretable, empirically grounded guidance. These heuristics are not designed for systematic enforcement in day-to-day development but rather to complement existing practices, especially in environments lacking formal guidelines. They can also serve as a foundation for future guidelines or assist in validating outputs from automated tools. These approaches are not mutually exclusive and may benefit from being combined in future tools, where automated suggestions can be checked against human-understandable logging principles.

In summary, the Prescriptive Phase of our study produced a comprehensive set of heuristics that address the specifics and the broader principles of LSL classification.

8. Related work

In this section, we review previous studies that explored software logging practices and proposed automated solutions for enhancing logging while considering the context of log severity levels.

8.1. Logging practices

Previous studies on logging practices have primarily focused on the modifications and maintenance associated with logging, without delving deeply into the motivations behind developers' choices regarding LSL adjustments.

Yuan et al. (2012a) analyzed four C/C++ systems and found that developers spend significant effort adjusting LSL. The study noted that 26% of log improvements involve severity level changes, and in 28% of these modifications, developers reconsider the trade-offs between different verbosity levels, indicating confusion about evaluating the costs (e.g., excessive messages, overhead) and benefits of each severity level. Chen and Jiang (2017b) replicated this study using Java systems classified severity level adjustments into two categories: *error-level updates* (changes to or from *error-levels* for *Failure purposes*) and *non-error-level updates* (neither the previous nor the current severity levels are errors). Building on these findings, Zeng et al. (2019) also replicated Yuan et al. (2012a)'s study for Android applications, finding less frequent severity adjustments compared to previous studies. Additionally, adjustments in the *error-level category* were less common, aligning with Chen and Jiang (2017b)'s findings. These studies demonstrate that, while adjustments are common, the underlying reasons behind these changes are not always well understood.

Li et al. (2020a) investigated the trade-offs reported in Yuan et al. (2012a)'s study by analyzing 223 issue reports and surveying 66 developers. They concluded that developers often balance the costs and benefits of logging by appropriately assigning severity levels. Building on these insights, our study delves deeper into the reasons behind these severity level adjustments, using issue reports to explore the specific motivations for these changes.

Li et al. (2017b) identify four categories for log changes, subdivided into 20 reasons, with only one addressing inappropriate severity levels, highlighting that a more detailed understanding of severity adjustments remains limited. Similarly, Zhang et al. (2022) investigated differences in logging characteristics between production and test environments, conducting a quantitative analysis of severity levels in both contexts. Li et al. (2021c) provide a broader discussion on severity levels, focusing on logging exception stack traces. They identify how developers log and modify exceptions, including severity levels and changes to those levels. The study suggests that exception stack traces should be avoided or logged at low levels for user errors, normal executions, and expected exceptions, emphasizing the importance of guidelines for logging, including severity levels. Their work emphasizes the importance of guidelines for logging practices; we investigate severity level changes across various contexts and explore the specific reasons behind these adjustments, complementing the guidelines they suggested and applying them to a broader range of logging scenarios.

Similarly, Patel et al. (2022) studied logging practices in the Linux kernel, observing changes in log statements across 22 releases, including severity adjustments. However, they did not explore the underlying causes of these changes, attributing adjustments solely to the difficulty in deciding the appropriate severity level. Our study not only addresses this but also expands on it by analyzing the reasons behind these adjustments, as documented in issue reports.

In terms of addressing logging-related issues, Chen and Jiang (2017a) and Hassani et al. (2018), identified several problems in open-source projects, including inappropriate log levels and proposed automated solutions to detect and correct mismatches between log messages and their severity levels. Chen and Jiang (2017a) further analyzed logging code changes in three open-source systems and identified six anti-patterns, including incorrect severity levels, underscoring the need for more specific guidelines for associating information with severity levels.

While previous studies have provided valuable insights into developers' intentions and practices when adjusting severity levels, they have not thoroughly explored the reasons behind these adjustments

or provided a detailed categorization of them. Our research fills this gap by proposing a comprehensive framework for understanding and applying severity levels. Unlike studies that focus on specific scenarios, such as exception stack traces in Java systems, our work categorizes developers' intentions in changing severity levels across various situations and provides practical heuristics to guide these decisions.

8.2. Automated approaches

Previous research on automated approaches for managing LSL has focused on developing tools and models to recommend, validate, and correct log severity decisions, but has often overlooked the underlying motivations behind these adjustments, which are crucial for improving manual logging practices.

Yuan et al. (2012a) developed a verbosity level checker to identify inconsistencies in severity levels across similar code snippets, detecting code clones and ensuring that their logging code maintained consistent verbosity. Chen and Jiang (2017a) introduced LCAnalyzer, a static code analysis tool designed to detect anti-patterns, including incorrect verbosity levels in logging practices. Building on these automated solutions, Li et al. (2017a) proposed an approach to automatically recommend appropriate severity levels for new log statements. Their method analyzes contextual features such as log churn, dynamic variable counts, and static text length, ensuring consistency with existing logging practices within a project. In a similar vein, Anu et al. (2019) proposed an automatic approach to assist developers in distinguishing ambiguous severity levels by extracting contextual features from logging code snippets and using a machine learning model to predict the appropriate level.

Expanding on this, Kim et al. (2020) developed an approach to validate and recommend log levels based on semantic and syntactic features, using feature vectors to quantify similarities among log messages and their surrounding code. Li et al. (2021a) analyzed inconsistencies in severity levels in duplicate log statements, presenting an automated tool to detect and correct them. Likewise, Li et al. (2021b) proposed DeepLV, a deep learning-based approach to suggest severity levels using syntactic context and log message attributes, though this work does not explain why levels like *Trace*, *Debug*, and *Warn* describe certain operations. Tang et al. (2022) introduced a model that adjusts severity levels based on the Mylyn DOI model, which adapts the severity according to the perceived relevance of the surrounding code. In a related study, Liu et al. (2022) proposed TeLL, an approach that uses multi-level block information to predict LSL. More recently, Xu et al. (2024) introduced UniLog, an LLM-based framework that predicts logging positions, severity levels, and messages. Evaluated on over 12,000 code snippets, UniLog achieved 72.3% accuracy for severity level prediction, illustrating the growing use of large language models in logging automation.

While these studies focus on developing automated tools to recommend, validate, or correct LSL, our study takes a different approach by focusing on the developers' intentions behind log severity adjustments. Instead of automating the process, we analyzed issue reports to investigate the reasons developers make these changes and distilled this understanding into 24 heuristics. These heuristics provide developers with structured guidelines for making more informed decisions regarding LSL.

By broadening the scope beyond specific automated tools and incorporating a variety of logging scenarios our study contributes significantly to improving the practical application of logging practices. This human-centered approach complements existing automated solutions by offering a structured framework that helps developers make informed decisions about LSL in real-world scenarios.

9. Threats to validity

It is important to acknowledge that there will always be potential threats to the validity of a study. However, we work carefully to identify and address eventual issues that could compromise the validity of our results or conclusions. In the following discussion, we outline the main threats to the validity of our study and describe the strategies we have implemented to mitigate them.

9.1. Internal validity

Threats to internal validity are concerned with the rigor (and thus the degree of control) of the study design.

Log statement selection method. The detection of log statements using SLogAnalyzer employs regular expressions; therefore, some of them may be missing in the selected set used for each system. To mitigate this problem, SLogAnalyzer makes use of a well-defined pipeline for extracting instructions, including cleaning non-relevant data, capturing the structure of each source file, identifying the maximum code blocks present, and only then applying a regular expression that identifies the log statements in the detected blocks.

Adjustments tracking. Comparison of logging statements between releases may have incorrectly identified log severity adjustments by associating non-matching code snippets. However, the SLogAnalyzer pipeline identifies the blocks in which each log instruction is located, comparing the instructions and the blocks that contain them. Additionally, we manually validated the adjustments associated with the issues used for our analyses to increase validity.

Issues selection method. Issues were also filtered using a regular expression, potentially missing some relevant issues concerning severity level adjustments or logging practices. Prior to this approach, we tested some techniques based on three machine learning algorithms on an initial set and analyzed the results. In this case, regular expression proved to be more efficient when selecting a larger group of issues for subsequent analysis, which increases the validity of the results.

Stable releases selection. We limited our study to stable releases of the selected systems, which may have caused us to miss some trends regarding log severity adjustments in "intra-release" adjustments. To mitigate this problem, inspired by the backward snowballing method of systematic literary reviews (Keele et al., 2007), we also analyzed the issues cited in the descriptions and comments of the selected issues.

Bias in data analysis. The process of interpreting Jira issue text is inherently subjective and can lead to biased results. To mitigate this, we adopted a systematic approach to data analysis. Issues were analyzed independently by the authors R1 e R2, to reduce bias. After discussions to converge interpretations, the results were revisited to exclude potential biases, thus increase validity. Additionally, all issues were rigorously evaluated by at least two authors.

Reliability of data sources. The accuracy and completeness of issue descriptions and comments in Jira can significantly influence our findings. There is a risk that misinterpretations may arise due to incomplete or misleading descriptions of issues. To mitigate this problem, we excluded from our evaluation issues that lacked explanations about the severity adjustment, despite the title being explicitly about the topic.

Bias in heuristic selection. The internal validity of this study may be influenced by methodological limitations, particularly in the selection of heuristics. This process relied on the interpretation of data from incident reports and logs, which could have introduced interpretative bias. While the heuristics were derived from recurring observations and consistent pattern analysis, certain severity classification decisions may reflect contexts specific to the analyzed projects. However, in the three systems studied, each commit is associated with an incident report. Therefore, we expect that the heuristics derived from these reports accurately represent the developers' intentions when adjusting LSL.

Generalization of findings. Another threat to internal validity is the generalization of observations drawn from a subset of projects. Although the set of open-source projects analyzed is well-established, the diversity of practices and maturity levels across systems may have influenced the derivation of heuristics. In cases where severity adjustments reflect individual preferences or informal organizational policies, the recommendations may not fully generalize to similar systems. To address this limitation, the heuristic construction followed an iterative process that combined multiple perspectives from the reports, including potential heuristics and preliminary observations.

9.2. External validity

Threats to external validity are any factors within a study that reduce the generality of the results.

Diversity of logging libraries. We focus on a limited range of logging libraries, which may not represent the full spectrum of logging practices. This limitation could affect the accuracy of our severity level categorization. However, we employed a methodology to select a representative library set and applied validated criteria for inclusion, enhancing the validity of our results.

Diversity of selected systems. Our findings are based on a specific set of open-source Java systems, which may not represent other software systems or programming languages. Our objective is to understand the state of practice based on a significant set of closed system releases, and generate heuristics that can contribute as guides in the logging process. It is not our goal to find results that can be generalized to all LSL choice situations. Furthermore, we apply validated and well-defined inclusion and exclusion criteria, selecting only closed issues to increase the validity of our results. In future work, we will confirm our conclusions by researching a wide range of systems, and therefore, issue texts.

Applicability across development contexts. Our analysis focused on specific open-source systems maintained by a single organization, which may not translate directly to enterprise environments or closed-source systems. Developers from other software systems or organizations may follow different logging practices and considerations. Additionally, severity levels and their nomenclature can vary significantly between logging libraries and frameworks, requiring adaptations of the heuristics to the specifics of these environments. This limited project scope may restrict the generalizability of our results to other systems.

For example, while mobile applications follow structured logging models, their practices are influenced by platform constraints such as energy and privacy (Zeng et al., 2019). Similarly, a study conducted on logging practices in proprietary software systems highlighted challenges similar to those found in open-source projects, such as inconsistent use of severity levels and balancing verbosity with usefulness (Fu et al., 2014). These findings suggest that our heuristics may be relevant beyond open-source environments, although adaptations may be required to address domain-specific constraints.

In addition, the logging model targeted in our study, *i.e.,* structured logging systems with explicit severity levels, is widely adopted. As discussed in Section 2.2.1, despite the variation in severity level nomenclature, we observed a trend toward convergence around six levels: *Trace*, *Debug*, *Info*, *Warn*, *Error*, and *Fatal*. This consistency suggests that our heuristics may be applicable to other systems that follow similar conventions.

That said, certain types of logs, such as those used for security auditing or low-level monitoring, may not incorporate hierarchical severity levels. In such cases, the direct application of our heuristics is limited.

The three open-source projects studied, although actively developed and maintained, do not necessarily guarantee full compliance with best logging practices. However, the decision to focus on resolved incident reports mitigates this bias by providing a more reliable perspective on severity adjustments successfully applied and validated by experienced developers. Furthermore, these projects are widely used in the industry as components of solutions developed by other companies, imposing high-quality requirements on the systems analyzed. These characteristics increase the credibility of our findings, although further validation in additional domains remains necessary.

9.3. Construct validity

Threats to construct validity refer to the degree to which the constructs used in the study truly represent what is intended to be measured.

Severity level and adjustment constructs. This concerns whether the severity adjustments and their motivations, as analyzed from issue reports and commits, accurately reflect the developers' rationale. To mitigate this risk, we grounded our interpretation of severity levels in definitions established through a previous systematic review and a mapping study of 40 logging libraries. These served as the conceptual basis for our classification and analysis throughout the study.

Impact of incorrect initial severity levels. Another concern is the assumption that severity levels present in each release accurately reflect developers' original intent. In reality, several issue reports in our dataset describe severity levels that were later acknowledged as inadequate, prompting an adjustment. This raises the possibility that some of our heuristics may reflect corrective patterns rather than ideal classification strategies. In other words, our analysis may capture how developers respond to misclassifications, not how they would assign severity correctly from the beginning. We consider this a threat to construct validity. Nonetheless, by identifying recurring justifications across many cases, *i.e.,* such as reducing noise or increasing clarity, we aim to derive heuristics that anticipate and prevent these misclassifications in future practice.

9.4. Conclusion validity

Threats to conclusion validity relate to the extent to which the results and interpretations are credible and justifiable.

Use of resolved issue reports. To mitigate this risk, we limited our analysis to adjustments associated with resolved issue reports. This approach ensures that the studied severity level changes reflect actual decisions made and validated by developers in response to real scenarios. These reports provide justifications for severity changes, reducing the likelihood of misinterpreting speculative or temporary edits.

Iterative analysis and researcher triangulation. The derivation of heuristics followed an iterative process involving multiple researchers, combining frequency analysis with qualitative validation of issue content. This approach ensured that interpretations were not based on a single perspective, increasing the robustness of the findings.

Severity level accuracy in release versions. We acknowledge that some log severity levels present in release versions may have been incorrectly assigned. However, our methodology focused on the subsequent adjustments justified in issue reports, which helps mitigate this risk. These adjustments reflect informed corrections by developers and offer a more reliable foundation for deriving heuristics.

Limitations of generalizability. Our conclusions are based on patterns observed in three open-source projects and should not be interpreted as statistical generalizations. Because we did not conduct controlled experiments, the observed relationships, such as the dominance of experience, driven adjustments or the prevalence of 1-degree adjustments, should be considered contextual insights rather than causal claims. Furthermore, the effectiveness of the proposed heuristics remains theoretical at this stage and will require future empirical validation in real-world settings.

10. Conclusion

In production software systems, logs are crucial for developers and operations engineers to diagnose system behavior and investigate failures. In this context, choosing log severity level (LSL) is fundamental to determining the relevance and visibility of log entries in development and production environments.

In this study, we proposed a three-phase methodology to investigate LSL based on severity adjustments observed in open-source projects. Our investigation of source code and associated issue reports supports the concept of logging's four primary purposes: *Debugging*, *Informational*, *Warning*, and *Failure*. These adjustments were grouped into categories reflecting distinct motivations, such as fundamental principles, historical practices, developer experience, and guideline adherence.

Our findings highlight the prevalence of experience-driven adjustments, the dominance of 1-degree adjustments (notably between *Debug* and *Info*), the need to reduce log verbosity in production, and the mutability of severity levels based on system maturity.

From our observations, we derived a set of 24 heuristics to guide the classification, review, and adjustment of severity levels. While not exhaustive, we believe these heuristics provide a flexible yet structured foundation for interpreting log messages according to their severity level. They are grounded in real-world practices observed across three mature Apache projects (Hadoop, HBase, Kafka), but we caution that these results may not generalize to all software systems. Projects from different domains (*e.g.,* web applications, embedded systems) or using other logging frameworks (*e.g.,* Python or .NET) may exhibit different logging behaviors. As such, our heuristics should be seen as a hypothesis of best practices that requires validation in broader contexts.

In future work, we are committed to rigorously evaluating the practicality and effectiveness of these heuristics. We plan to apply these heuristics in real-world scenarios and in a survey with software developers and operators, and in a controlled experiment. This experiment would compare severity level classifications performed with and without the heuristics, allowing us to assess their impact on consistency and correctness. We also consider their potential integration into static analysis tools to support automated review of log statements or continuous integration pipelines. Although this would require formalizing the heuristics into machine-readable rules and addressing the inherent contextual nuances of logging. Furthermore, future work should include the analysis of additional systems, particularly from different organizations and domains, to assess whether the observed severity adjustment patterns and proposed heuristics generalize beyond the three Apache projects studied. Another route for future work involves developing a formal theory of severity levels, considering assigning a level to a log statement as an optimization problem. This approach aims to optimize the balance between the informativeness of log data and resource constraints, such as storage and processing overhead.

Through these efforts, we aim to contribute to a broader understanding of log management in software systems, offering tools and theoretical knowledge that improve the diagnostic capabilities of developers and system operators. By bridging the gap between theory and practice in logging, our work strives to pave the way for more efficient and effective log analysis strategies in the ever-evolving software development landscape.

CRediT authorship contribution statement

Eduardo Mendes: Writing – original draft, Visualization, Software, Project administration, Methodology, Investigation, Data curation, Conceptualization. **Marcelo Vasconcellos:** Validation, Software, Methodology, Investigation. **Fabio Petrillo:** Conceptualization. **Sylvain Hallé:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization.

Declaration of competing interest

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

Acknowledgments

This project has received funding from Canada Research Chairs (CRC-2020-00308) and Natural Sciences and Engineering Research Council of Canada (RGPIN-2023-05271). The authors thank the anonymous reviewers for their genuine interest and for the many constructive and insightful comments that greatly improved this manuscript.

Appendix A. Initial tags for log severity adjustments

The table below lists the 45 descriptive tags used in the initial stage of our analysis. These labels summarize the motivations extracted from issue reports concerning log severity level changes.

• Bad adjustment	• INFO not justified
• Benign message at WARN level	• Large volume of logs
• Critical information to troubleshooting	• Library swapping
• Data absence to DEBUG	• Masking DEBUG into an INFO
• Debug masked as INFO	• Misconception about error level
• Details at INFO level	• Noise
• Disabled feature	• Non-existent level
• Discussion about another level	• Normally behavior at WARN level
• Don't explain about the adjustment reason	• Not affect the normal service
• Event handled by client	• Not a frequently log message
• Event is not quite right but doesn't require any action	• Not related to the adjustment
• Excessive logging	• Potential Verbose INFO
• Expensive INFO	• Silent problem
• Flooding log	• Spammy logging
• General Guideline	• Swapping library
• Heuristic	• Temporary INFO
• Ignored interruptions at WARN	• Verbose DEBUG
• Inconsistency	• Verbose ERROR
• Inconsistent practice	• Verbose WARN
• Inconsistent log practice	• Verbose INFO
• INFO classified as DEBUG	• Weak information
• INFO level for analysis purpose	

These tags were grouped into intermediate categories, which in turn served as the basis for the five main categories described in Section 3.3.2.

Appendix B. Hadoop issues

See Table B.1.

Appendix C. HBase issues

See Table C.1.

Appendix D. Kafka issues

See Table D.1.

Table B.1
Hadoop issues.

#	ID	URL	Summary
#1	HADOOP-96	https://issues.apache.org/jira/browse/HADOOP-96	name server should log decisions that affect data: block creation, removal, replication
#2	HADOOP-211	https://issues.apache.org/jira/browse/HADOOP-211	logging improvements for Hadoop
#3	HADOOP-1034	https://issues.apache.org/jira/browse/HADOOP-1034	RuntimeException and Error not catched in DataNode.DataXceiver.run()
#4	HADOOP-7858	https://issues.apache.org/jira/browse/HADOOP-7858	Drop some info logging to DEBUG level in IPC, metrics, and HTTP
#5	HADOOP-8075	https://issues.apache.org/jira/browse/HADOOP-8075	Lower native-hadoop library log from info to debug
#6	HADOOP-8932	https://issues.apache.org/jira/browse/HADOOP-8932	JNI-based user-group mapping modules can be too chatty on lookup failures
#7	HADOOP-9135	https://issues.apache.org/jira/browse/HADOOP-9135	JniBasedUnixGroupsMappingWithFallback should log at debug rather than info during fallback
#8	HADOOP-9582	https://issues.apache.org/jira/browse/HADOOP-9582	Non-existent file to "hadoop fs -conf" doesn't throw error
#9	HADOOP-10015	https://issues.apache.org/jira/browse/HADOOP-10015	UserGroupInformation prints out excessive ERROR warnings
#10	HADOOP-10274	https://issues.apache.org/jira/browse/HADOOP-10274	Lower the logging level from ERROR to WARN for UGI.doAs method
#11	HADOOP-10343	https://issues.apache.org/jira/browse/HADOOP-10343	Change info to debug log in LossyRetryInvocationHandler
#12	HADOOP-10466	https://issues.apache.org/jira/browse/HADOOP-10466	Lower the log level in UserGroupInformation
#13	HADOOP-10657	https://issues.apache.org/jira/browse/HADOOP-10657	Have RetryInvocationHandler log failover attempt at INFO level
#14	HADOOP-11085	https://issues.apache.org/jira/browse/HADOOP-11085	Excessive logging by org.apache.hadoop.util.Progress when value is NaN
#15	HADOOP-12789	https://issues.apache.org/jira/browse/HADOOP-12789	log classpath of ApplicationClassLoader at INFO level
#16	HADOOP-13552	https://issues.apache.org/jira/browse/HADOOP-13552	RetryInvocationHandler logs all remote exceptions
#17	HADOOP-14539	https://issues.apache.org/jira/browse/HADOOP-14539	Move commons logging APIs over to slf4j in hadoop-common
#18	HADOOP-14987	https://issues.apache.org/jira/browse/HADOOP-14987	Improve KMSClientProvider log around delegation token checking
#19	HADOOP-15441	https://issues.apache.org/jira/browse/HADOOP-15441	Log kms url and token service at debug level.
#20	HADOOP-15942	https://issues.apache.org/jira/browse/HADOOP-15942	Change the logging level form DEBUG to ERROR for RuntimeErrorException in JMXJsonServlet
#21	HADOOP-17597	https://issues.apache.org/jira/browse/HADOOP-17597	Add option to downgrade S3A rejection of Syncable to warning
#22	HADOOP-17836	https://issues.apache.org/jira/browse/HADOOP-17836	Improve logging on ABFS error reporting
#23	HADOOP-18065	https://issues.apache.org/jira/browse/HADOOP-18065	ExecutorHelper.logThrowableFromAfterExecute() is too noisy.
#24	HADOOP-18574	https://issues.apache.org/jira/browse/HADOOP-18574	Changing log level of IOStatistics increment to make the DEBUG logs less noisy
#25	HDFS-6085	https://issues.apache.org/jira/browse/HDFS-6085	Improve CacheReplicationMonitor log messages a bit
#26	HDFS-6998	https://issues.apache.org/jira/browse/HDFS-6998	warning message 'ssl.client.truststore.location has not been set' gets printed for hftp command
#27	HDFS-8659	https://issues.apache.org/jira/browse/HDFS-8659	Block scanner INFO message is spamming logs
#28	HDFS-9906	https://issues.apache.org/jira/browse/HDFS-9906	Remove spammy log spew when a datanode is restarted
#29	HDFS-10377	https://issues.apache.org/jira/browse/HDFS-10377	CacheReplicationMonitor shutdown log message should use INFO level.
#30	HDFS-10752	https://issues.apache.org/jira/browse/HDFS-10752	Several log refactoring/improvement suggestion in HDFS
#31	HDFS-11054	https://issues.apache.org/jira/browse/HDFS-11054	Suppress verbose log message in BlockPlacementPolicyDefault
#32	HDFS-11593	https://issues.apache.org/jira/browse/HDFS-11593	Change SimpleHttpProxyHandler#exceptionCaught log level from info to debug
#33	HDFS-13692	https://issues.apache.org/jira/browse/HDFS-13692	StorageInfoDefragmenter floods log when compacting StorageInfo TreeSet
#34	HDFS-13695	https://issues.apache.org/jira/browse/HDFS-13695	Move logging to slf4j in HDFS package
#35	HDFS-14238	https://issues.apache.org/jira/browse/HDFS-14238	A log in NNThroughputBenchmark should change log level to "INFO" instead of "ERROR"
#36	HDFS-14339	https://issues.apache.org/jira/browse/HDFS-14339	Inconsistent log level practices in RpcProgramNfs3.java
#37	HDFS-14340	https://issues.apache.org/jira/browse/HDFS-14340	Lower the log level when can't get postOpAttr
#38	HDFS-14395	https://issues.apache.org/jira/browse/HDFS-14395	Remove WARN Logging From Interrupts in DataStreamer
#39	HDFS-14521	https://issues.apache.org/jira/browse/HDFS-14521	Suppress setReplication logging.
#40	HDFS-14759	https://issues.apache.org/jira/browse/HDFS-14759	HDFS cat logs an info message
#41	HDFS-14760	https://issues.apache.org/jira/browse/HDFS-14760	Log INFO mode if snapshot usage and actual usage differ
#42	HDFS-15197	https://issues.apache.org/jira/browse/HDFS-15197	[SBN read] Change ObserverRetryOnActiveException log to debug
#43	MAPREDUCE-3184	https://issues.apache.org/jira/browse/MAPREDUCE-3184	Improve handling of fetch failures when a tasktracker is not responding on HTTP
#44	MAPREDUCE-3265	https://issues.apache.org/jira/browse/MAPREDUCE-3265	Reduce log level on MR2 IPC construction, etc
#45	MAPREDUCE-3348	https://issues.apache.org/jira/browse/MAPREDUCE-3348	mapred job -status fails to give info even if the job is present in History
#46	MAPREDUCE-3692	https://issues.apache.org/jira/browse/MAPREDUCE-3692	yarn-resourcemanager out and log files can get big
#47	MAPREDUCE-3748	https://issues.apache.org/jira/browse/MAPREDUCE-3748	Move CS related nodeUpdate log messages to DEBUG
#48	MAPREDUCE-4570	https://issues.apache.org/jira/browse/MAPREDUCE-4570	ProcfsBasedProcessTree#constructProcessInfo() prints a warning if procfsDir/<pid>/stat is not found.
#49	MAPREDUCE-4614	https://issues.apache.org/jira/browse/MAPREDUCE-4614	Simplify debugging a job's tokens
#50	MAPREDUCE-5766	https://issues.apache.org/jira/browse/MAPREDUCE-5766	Ping messages from attempts should be moved to DEBUG
#51	MAPREDUCE-6971	https://issues.apache.org/jira/browse/MAPREDUCE-6971	Moving logging APIs over to slf4j in hadoop-mapreduce-client-app
#52	MAPREDUCE-6983	https://issues.apache.org/jira/browse/MAPREDUCE-6983	Moving logging APIs over to slf4j in hadoop-mapreduce-client-core
#53	MAPREDUCE-6997	https://issues.apache.org/jira/browse/MAPREDUCE-6997	Moving logging APIs over to slf4j in hadoop-mapreduce-client-hs
#54	MAPREDUCE-6998	https://issues.apache.org/jira/browse/MAPREDUCE-6998	Moving logging APIs over to slf4j in hadoop-mapreduce-client-jobclient
#55	MAPREDUCE-7022	https://issues.apache.org/jira/browse/MAPREDUCE-7022	Fast fail rogue jobs based on task scratch dir size
#56	MAPREDUCE-7063	https://issues.apache.org/jira/browse/MAPREDUCE-7063	Fix log level inconsistency in CombineFileInputFormat.java
#57	YARN-1022	https://issues.apache.org/jira/browse/YARN-1022	Unnecessary INFO logs in AMRMClientAsync
#58	YARN-1608	https://issues.apache.org/jira/browse/YARN-1608	LinuxContainerExecutor has a few DEBUG messages at INFO level
#59	YARN-1839	https://issues.apache.org/jira/browse/YARN-1839	Capacity scheduler preempts an AM out. AM attempt 2 fails to launch task container with SecretManager$InvalidToken: No NMToken sent
#60	YARN-1892	https://issues.apache.org/jira/browse/YARN-1892	Excessive logging in RM
#61	YARN-2213	https://issues.apache.org/jira/browse/YARN-2213	Change proxy-user cookie log in AmIpFilter to DEBUG
#62	YARN-2704	https://issues.apache.org/jira/browse/YARN-2704	Localization and log-aggregation will fail if hdfs delegation token expired after token-max-life-time
#63	YARN-3350	https://issues.apache.org/jira/browse/YARN-3350	YARN RackResolver spams logs with messages at info level
#64	YARN-4115	https://issues.apache.org/jira/browse/YARN-4115	Reduce loglevel of ContainerManagementProtocolProxy to Debug
#65	YARN-5693	https://issues.apache.org/jira/browse/YARN-5693	Reduce loglevel to Debug in ContainerManagementProtocolProxy and AMRMClientImpl
#66	YARN-6068	https://issues.apache.org/jira/browse/YARN-6068	Log aggregation get failed when NM restart even with recovery
#67	YARN-6873	https://issues.apache.org/jira/browse/YARN-6873	Moving logging APIs over to slf4j in hadoop-yarn-server-applicationhistoryservice
#68	YARN-6957	https://issues.apache.org/jira/browse/YARN-6957	Moving logging APIs over to slf4j in hadoop-yarn-server-sharedcachemanager
#69	YARN-7047	https://issues.apache.org/jira/browse/YARN-7047	Moving logging APIs over to slf4j in hadoop-yarn-server-nodemanager
#70	YARN-7243	https://issues.apache.org/jira/browse/YARN-7243	Moving logging APIs over to slf4j in hadoop-yarn-server-resourcemanager
#71	YARN-7407	https://issues.apache.org/jira/browse/YARN-7407	Moving logging APIs over to slf4j in hadoop-yarn-applications
#72	YARN-7477	https://issues.apache.org/jira/browse/YARN-7477	Moving logging APIs over to slf4j in hadoop-yarn-common
#73	YARN-7727	https://issues.apache.org/jira/browse/YARN-7727	Incorrect log levels in few logs with QueuePriorityContainerCandidateSelector
#74	YARN-8459	https://issues.apache.org/jira/browse/YARN-8459	Improve Capacity Scheduler logs to debug invalid states
#75	YARN-9349	https://issues.apache.org/jira/browse/YARN-9349	When doTransition() method occurs exception, the log level practices are inconsistent
#76	YARN-10369	https://issues.apache.org/jira/browse/YARN-10369	Make NMTokenSecretManagerInRM sending NMToken for nodeId DEBUG
#77	YARN-10997	https://issues.apache.org/jira/browse/YARN-10997	Revisit allocation and reservation logging

Table C.1
HBase issues.

#	ID	URL	Summary
#1	HBASE-5582	https://issues.apache.org/jira/browse/HBASE-5582	"No HServerInfo found for" should be a WARNING message
#2	HBASE-6023	https://issues.apache.org/jira/browse/HBASE-6023	Normalize security audit logging level with Hadoop
#3	HBASE-7037	https://issues.apache.org/jira/browse/HBASE-7037	ReplicationPeer logs at WARN level aborting server instead of at FATAL
#4	HBASE-7214	https://issues.apache.org/jira/browse/HBASE-7214	CleanerChore logs too much, so much so it obscures all else that is going on
#5	HBASE-8940	https://issues.apache.org/jira/browse/HBASE-8940	TestRegionMergeTransactionOnCluster#testWholesomeMerge may fail due to race in opening region
#6	HBASE-9120	https://issues.apache.org/jira/browse/HBASE-9120	ClassFinder logs errors that are not
#7	HBASE-9371	https://issues.apache.org/jira/browse/HBASE-9371	Eliminate log spam when tailing files
#8	HBASE-10092	https://issues.apache.org/jira/browse/HBASE-10092	Move to slf4j
#9	HBASE-10906	https://issues.apache.org/jira/browse/HBASE-10906	Change error log for NamingException in TableInputFormatBase to WARN level
#10	HBASE-12419	https://issues.apache.org/jira/browse/HBASE-12419	"Partial cell read caused by EOF" ERRORs on replication source during replication
#11	HBASE-12461	https://issues.apache.org/jira/browse/HBASE-12461	FSVisitor logging is excessive
#12	HBASE-12539	https://issues.apache.org/jira/browse/HBASE-12539	HFileLinkCleaner logs are uselessly noisy
#13	HBASE-13675	https://issues.apache.org/jira/browse/HBASE-13675	ProcedureExecutor completion report should be at DEBUG log level
#14	HBASE-14042	https://issues.apache.org/jira/browse/HBASE-14042	Fix FATAL level logging in FSHLog where logged for non fatal exceptions
#15	HBASE-15582	https://issues.apache.org/jira/browse/HBASE-15582	SnapshotManifestV1 too verbose when there are no regions
#16	HBASE-15954	https://issues.apache.org/jira/browse/HBASE-15954	REST server should log requests with TRACE instead of DEBUG
#17	HBASE-16220	https://issues.apache.org/jira/browse/HBASE-16220	Demote log level for "HRegionFileSystem - No StoreFiles for" messages to TRACE
#18	HBASE-17540	https://issues.apache.org/jira/browse/HBASE-17540	Change SASL server GSSAPI callback log line from DEBUG to TRACE in RegionServer to reduce log volumes in DEBUG mode
#19	HBASE-20447	https://issues.apache.org/jira/browse/HBASE-20447	Only fail cacheBlock if block collisions aren't related to next block metadata
#20	HBASE-20554	https://issues.apache.org/jira/browse/HBASE-20554	"WALs outstanding" message from CleanerChore is noisy
#21	HBASE-20665	https://issues.apache.org/jira/browse/HBASE-20665	"Already cached block XXX" message should be DEBUG
#22	HBASE-20701	https://issues.apache.org/jira/browse/HBASE-20701	too much logging when balancer runs from BaseLoadBalancer
#23	HBASE-20770	https://issues.apache.org/jira/browse/HBASE-20770	WAL cleaner logs way too much; gets clogged when lots of work to do
#24	HBASE-21524	https://issues.apache.org/jira/browse/HBASE-21524	Unnecessary DEBUG log in ConnectionImplementation#isTableEnabled
#25	HBASE-23047	https://issues.apache.org/jira/browse/HBASE-23047	ChecksumUtil.validateChecksum logs an INFO message inside a "if(LOG.isTraceEnabled())" block.
#26	HBASE-23250	https://issues.apache.org/jira/browse/HBASE-23250	Log message about CleanerChore delegate initialization should be at INFO
#27	HBASE-23687	https://issues.apache.org/jira/browse/HBASE-23687	DEBUG logging cleanup
#28	HBASE-24524	https://issues.apache.org/jira/browse/HBASE-24524	SyncTable logging improvements
#29	HBASE-25483	https://issues.apache.org/jira/browse/HBASE-25483	set the loadMeta log level to debug.
#30	HBASE-25556	https://issues.apache.org/jira/browse/HBASE-25556	Frequent replication "Encountered a malformed edit" warnings
#31	HBASE-25642	https://issues.apache.org/jira/browse/HBASE-25642	Fix or stop warning about already cached block
#32	HBASE-26189	https://issues.apache.org/jira/browse/HBASE-26189	Reduce log level of CompactionProgress notice to DEBUG
#33	HBASE-26443	https://issues.apache.org/jira/browse/HBASE-26443	Some BaseLoadBalancer log lines should be at DEBUG level
#34	HBASE-27079	https://issues.apache.org/jira/browse/HBASE-27079	Lower some DEBUG level logs in ReplicationSourceWALReader to TRACE
#35	HBASE-27391	https://issues.apache.org/jira/browse/HBASE-27391	Downgrade ERROR log to DEBUG in ConnectionUtils.updateStats
#36	HBASE-27588	https://issues.apache.org/jira/browse/HBASE-27588	"Instantiating StoreFileTracker impl" INFO level logging is too chatty

Table D.1
Kafka issues.

#	ID	URL	Summary
#1	KAFKA-5704	https://issues.apache.org/jira/browse/KAFKA-5704	Auto topic creation causes failure with older clusters
#2	KAFKA-4829	https://issues.apache.org/jira/browse/KAFKA-4829	Improve logging of StreamTask commits
#3	KAFKA-6802	https://issues.apache.org/jira/browse/KAFKA-6802	Improve logging when topics aren't known and assignments skipped
#4	KAFKA-9540	https://issues.apache.org/jira/browse/KAFKA-9540	Application getting "Could not find the standby task 0_4 while closing it" error
#5	KAFKA-13037	https://issues.apache.org/jira/browse/KAFKA-13037	"Thread state is already PENDING_SHUTDOWN" log spam
#6	KAFKA-13669	https://issues.apache.org/jira/browse/KAFKA-13669	Log messages for source tasks with no offsets to commit are noisy and confusing

Data availability

The data and analytic code for this study are available on the FigShare repository, accessible at https://doi.org/10.6084/m9.figshare.26253776.v2.

References

Anu, H., Chen, J., Shi, W., Hou, J., Liang, B., Qin, B., 2019. An approach to recommendation of verbosity log levels based on logging intention. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 125–134.

Apache, 2006. Hadoop. URL: https://github.com/apache/hadoop.

Bradner, S., 1997. RFC2119: Key words for use in RFCs to indicate requirement levels.

Cândido, J., Aniche, M., van Deursen, A., 2021. Log-based software monitoring: a systematic mapping study. PeerJ Comput. Sci. 7, e489.

Chen, B., Jiang, Z.M., 2017a. Characterizing and detecting anti-patterns in the logging code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 71–81.

Chen, B., Jiang, Z.M.J., 2017b. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. Empir. Softw. Eng. 22 (1), 330–374.

Chowdhury, S., Di Nardo, S., Hindle, A., Jiang, Z.M.J., 2018. An exploratory study on assessing the energy impact of logging on android applications. Empir. Softw. Eng. 23 (3), 1422–1456.

Ding, R., Zhou, H., Lou, J.G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., Xie, T., 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In: 2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15). pp. 139–150.

El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A., Bouziane, A., 2020. A systematic literature review on automated log abstraction techniques. Inf. Softw. Technol. 122, 106276.

Farshchi, M., Schneider, J.G., Weber, I., Grundy, J., 2018. Metric selection and anomaly detection for cloud operations using log and metric correlation analysis. J. Syst. Softw. 137, 531–549.

Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., Xie, T., 2014. Where do developers log? an empirical study on logging practices in industry. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 24–33.

Glaser, B., Strauss, A., 2017. Discovery of Grounded Theory: Strategies for Qualitative Research. Routledge.

Hassani, M., Shang, W., Shihab, E., Tsantalis, N., 2018. Studying and detecting log-related issues. Empir. Softw. Eng. 23 (6), 3248–3280.

He, P., Chen, Z., He, S., Lyu, M.R., 2018. Characterizing the natural language descriptions in software logging statements. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 178–189.

Kabinna, S., Bezemer, C.P., Shang, W., Syer, M.D., Hassan, A.E., 2018. Examining the stability of logging statements. Empir. Softw. Eng. 23 (1), 290–333.

Keele, S., et al., 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report, Citeseer.

Kim, T., Kim, S., Park, S., Park, Y., 2020. Automatic recommendation to appropriate log levels. Softw.: Pr. Exp. 50 (3), 189–209.

Li, Z., Chen, T.H., Shang, W., 2020b. Where shall we log? studying and suggesting logging locations in code blocks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 361–372.

Li, H., Chen, T.H.P., Shang, W., Hassan, A.E., 2018. Studying software logging using topic models. Empir. Softw. Eng. 23 (5), 2655–2694.

Li, Z., Chen, T.H., Yang, J., Shang, W., 2021a. Studying duplicate logging statements and their relationships with code clones. IEEE Trans. Softw. Eng. 48 (7), 2476–2494.

Li, Z., Li, H., Chen, T.H., Shang, W., 2021b. Deeplv: Suggesting log levels using ordinal based neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 1461–1472.

Li, H., Shang, W., Adams, B., Sayagh, M., Hassan, A.E., 2020a. A qualitative study of the benefits and costs of logging from developers' perspectives. IEEE Trans. Softw. Eng. 47 (12), 2858–2873.

Li, H., Shang, W., Hassan, A.E., 2017a. Which log level should developers choose for a new logging statement? Empir. Softw. Eng. 22 (4), 1684–1716.

Li, H., Shang, W., Zou, Y., Hassan, A.E., 2017b. Towards just-in-time suggestions for log changes. Empir. Softw. Eng. 22 (4), 1831–1865.

Li, H., Zhang, H., Wang, S., Hassan, A.E., 2021c. Studying the practices of logging exception stack traces in open-source software projects. IEEE Trans. Softw. Eng. 48 (12), 4907–4924.

Lin, Q., Zhang, H., Lou, J.G., Zhang, Y., Chen, X., 2016. Log clustering based problem identification for online service systems. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion. ICSE-C, IEEE, pp. 102–111.

Liu, Z., Xia, X., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2019. Which variables should i log? IEEE Trans. Softw. Eng..

Liu, J., Zeng, J., Wang, X., Ji, K., Liang, Z., 2022. Tell: log level suggestions via modeling multi-level code block information. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 27–38.

M. Vasconcellos, M., 2023. Vers l'évaluation de la stabilité des systèmes à l'aide de la densité des logs (Master's thesis). Université du Québec à Chicoutimi.

Mendes, E., Petrillo, F., 2021. Log severity levels matter: A multivocal mapping. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 1002–1013.

Oracle, 2018. Throwable (Java Platform SE 8). URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html. (Accessed November 2023).

Patel, K., Faccin, J., Hamou-Lhadj, A., Nunes, I., 2022. The sense of logging in the linux kernel. Empir. Softw. Eng. 27 (6), 153.

Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M., 2008. Systematic mapping studies in software engineering. In: 12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12. pp. 1–10.

Plattel, C., 2014. Distributed and Incremental Clustering Using Shared Nearest Neighbours (Master's thesis). Utrecht University.

Rong, G., Gu, S., Zhang, H., Shao, D., Liu, W., 2018. How is logging practice implemented in open source software projects? a preliminary exploration. In: 2018 25th Australasian Software Engineering Conference. ASWEC, IEEE, pp. 171–180.

Shang, W., Nagappan, M., Hassan, A.E., 2015. Studying the relationship between logging characteristics and the code quality of platform software. Empir. Softw. Eng. 20 (1), 1–27.

Singhal, A., et al., 2001. Modern information retrieval: A brief overview. IEEE Data Eng. Bull. 24 (4), 35–43.

Tan, T., Steinbach, M., Kumar, V., 2005. Introduction to Data Mining. Pearson, Boston, MA.

Tang, Y., Spektor, A., Khatchadourian, R., Bagherzadeh, M., 2022. Automated evolution of feature logging statement levels using git histories and degree of interest. Sci. Comput. Program. 214, 102724.

Turney, P.D., Pantel, P., 2010. From frequency to meaning: Vector space models of semantics. J. Artificial Intelligence Res. 37, 141–188.

Xu, J., Cui, Z., Zhao, Y., Zhang, X., He, S., He, P., Li, L., Kang, Y., Lin, Q., Dang, Y., et al., 2024. UniLog: Automatic logging via LLM and in-context learning. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. pp. 1–12.

Yang, N., Schiffelers, R., Lukkien, J., 2021. An interview study of how developers use execution logs in embedded software engineering. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, IEEE, pp. 61–70. http://dx.doi.org/10.1109/icse-seip52600.2021.00015.

Yao, K., Li, H., Shang, W., Hassan, A.E., 2020. A study of the performance of general compressors on log files. Empir. Softw. Eng. 25 (5), 3043–3085.

Yuan, D., Park, S., Zhou, Y., 2012a. Characterizing logging practices in open-source software. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 102–112.

Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S., 2012b. Improving software diagnosability via log enhancement. ACM Trans. Comput. Syst. (TOCS) 30 (1), 1–28.

Zeng, Y., Chen, J., Shang, W., Chen, T.H.P., 2019. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. Empir. Softw. Eng. 24 (6), 3394–3434.

Zhang, H., Tang, Y., Lamothe, M., Li, H., Shang, W., 2022. Studying logging practice in test code. Empir. Softw. Eng. 27 (4), 83.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., Zhou, Y., 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 565–581.

Eduardo Mendes received his Ph.D. in Information Science and Technology from the University of Quebec at Chicoutimi (UQAC), Canada, in 2025. His doctoral research focused on logging theory, with a special emphasis on log severity levels. His research interests include software engineering, software architecture, and logging theory.

Marcelo Vasconcellos received his Master's degree in Computer Science from the Université du Québec à Chicoutimi (UQAC) in 2023. He is an enthusiast of Python (with over 10 years of experience), SQL databases, and other technologies. His expertise includes the analysis and development of systems, with focus on data science, artificial intelligence, big data, business intelligence, and ETL processes. His research interests include data science and software engineering.

Fabio Petrillo, received the Ph.D. degree in computer science from the Federal University of Rio Grande do Sul, Brazil, in 2016. He is an Associate Professor with the Department of Software Engineering Information Technology, École de Technologie Supérieure, Canada. He was a Post-doctoral Fellow at Concordia University, Canada. He has was a programmer, software architect, manager, and agile coach for more than 20 years, working on critical mission projects and guiding several teams. He has published several papers in international conferences and journals, including IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (TSE), European Master on Software Engineering (EMSE), Journal of Systems and Software (JSS), Information and Software Technology (IST), IEEE SOFTWARE, QRS, ICPC, SAC, ICSOC, and VISSOFT. His research interests include empirical software engineering, software quality, debugging, service-oriented architecture, cloud computing, and agile methods. He has been recognized as a Pioneer and an international reference on Software Engineering for Computer Games. He is the Creator of Swarm Debugging — a new collaborative approach to support debugging activities. He has served on the program committees of several international conferences, including QRS, CHI, SIGCSE, ICPC, VISSOFT, and GAS. He has reviewed for top international journals, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), Journal of Systems and Software (JSS), European Master on Software Engineering (EMSE), and Information and Software Technology (IST).

Sylvain Hallé, Ph.D., is a full professor in the Department of Computer Science and Mathematics at the University of Quebec at Chicoutimi, where he has worked since 2010. He is the current holder of the Canada Research Chair in Software Specification, Testing and Verification. He obtained a Ph.D. in computer science from the University of Quebec at Montreal in 2008, and pursued post-doctoral studies at the University of California at Santa Barbara from 2008 to 2010. Prof. Hallé's research interests include formal methods, event stream processing and the explainability of results produced by computer systems.